

前端撷英馆

揭秘 Angular



广发证券互联网金融技术团队 著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

作为一部系统讲解流行前端框架 Angular 新版的权威著作,本书覆盖入门、深入和实战三大主题。第一部分从前端的故事起点说起,然后对 Angular 及 TypeScript 进行了简单的介绍,接着通过一个通讯录例子让读者快速入门 Angular 的开发;第二部分深入讲解了 Angular 架构及 Angular 核心内容,包括组件、模板、指令、服务、依赖注入、路由及测试,此外,在相应的章节里还补充说明了如变化监测的核心 Zones(第6章)、双向绑定的原理(第7章)、RxJS(第9章)等关键内容;第三部分则通过问卷调查系统来指引读者进行 Angular 项目的实战;第四部分主要是 Angular 延伸知识的讲解,介绍了 ionic 框架(第19章)及 Angular 的服务端渲染(第20章)相关技术。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

揭秘 Angular / 广发证券互联网金融技术团队著. —2 版. —北京:电子工业出版社,2018.7
ISBN 978-7-121-34272-1

I. ①揭…II. ①广…III. ①超文本标记语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2018)第 109338 号

策划编辑:张春雨

责任编辑:葛 娜

印 刷:北京天宇星印刷厂

装 订:北京天宇星印刷厂

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱

邮编:100036

开 本:787×980 1/16 印张:36

字数:750 千字

版 次:2017 年 1 月第 1 版

2018 年 7 月第 2 版

印 次:2018 年 7 月第 1 次印刷

定 价:118.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888,88258888。

质量投诉请发邮件至 zltz@phei.com.cn,盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819 faq@phei.com.cn。

推荐序 1

让时间倒推至 2013 年年初。

这一年，笔者从科技外企、互联网界投入到本土金融机构，在所谓“互联网金融”的喧嚣中，开始招募与建设一个金融科技的研发组织。此刻正面临着一个个此前职业生涯所未遇的、因行业文化与组织管理差异而需重新学习适应的小“冲击”。其中印象最深的一个，无疑是在招聘过程中感受到来自互联网大企业的工程师对传统金融业 IT 的一定程度上的“蔑视”——通常在面试过程中面试者最正面的反应莫过于“嗨，没想到券商还有这样的技术”，更多的面试者则是把金融机构的前端技术想象停留在 Visual Basic、jQuery 阶段。传统金融机构，对于年轻的互联网技术人，很可能是恐龙般的存在。想想现在有多少年轻人从来不去银行营业部办业务，甚至从来不去商场购物？大部分人对于金融机构的印象，深受该机构的网站、手机 App 的影响。有很多面试者在面试中吐槽金融行业整体软件服务糟糕的用户体验，把面试变成一个尴尬的“用户反馈”渠道。这就是个人在转换行业后遭遇的难忘经历。

当然，这种经历也和我们“不自量力”，自找不痛快有关——作为“传统”IT 组织非得去“忽悠”顶级互联网企业的骄傲的牛人们加盟……可是，事实上很多传统行业的前端技术（以及很多其他技术）都没有与时俱进，如果我们不想办法建立起强悍的技术团队，采用更前沿、更领先的技术弯道超车，就永远无法解决用户体验糟糕的问题，永远难以改变在技术界留下的“技术落后”的负面印象，这些负分会让招募技术牛人更加困难，因为有追求的工程师都希望和一流的团队合作，开发有个人成就感的一流的技术应用。这是一个恶性循环，如果不打破它，我们既做不出好的产品，也招募不到好的工程师。

我们的办法是，采用“激进”的技术“破冰”。在 2013 年，类似于 Angular、Node.js、MongoDB、Docker 等这些技术，恐怕在大部分互联网公司里依然属于较为少用的技术，

更遑论以稳定可靠为导向的、技术路线保守的金融界。采用这些技术开发交易额动辄以“亿元”为单位的金融应用，可以称得上“激进”。可是我们有一个论调，就是：对于新兴技术有高度热情甚至“疯狂”的工程师，往往是技术比较强悍的工程师，他们驾驭故障、问题的能力，往往又是比较强的；相比之下，技术套路保守、比较求稳的团队，则不一定以技术见长。这两者权衡之下，我们决定冒一点风险，通过更前沿的技术吸引有技术热情的互联网人加盟，给他们玩新技术的自由，同时也考验他们填技术“坑”的能力。当然，这些技术在硅谷诞生已有时日，已经形成庞大的海外社区，我们并不是一拍脑袋为了“前沿”而“前沿”的，该做的论证依然需要严谨地做。最重要的一点是，我们首先应该纠正“盲目”的态度——对于任何新鲜技术，避免因为自己的不熟悉、缺乏调研就一概而论地斥之为“不成熟”，事实上，“没有调查研究就没有发言权”，抱着开放的心态深入了解新技术，会发现它们中有很多是从根本上经得起“思想实验”、符合逻辑、符合未来发展方向的。但我们可以看到，很多企业尤其是非科技行业的机构，往往对于任何新兴技术没有深入的、逻辑的、严谨科学的分析，而是简单粗暴地以技术存在的时间“年龄”作为采用与否的判断标准，这是令人遗憾的。

从 2013 年起，类似于 CoffeeScript、TypeScript、ES 6/7、Promise、Meteor.js、Yeoman、Ember.js、Babel、Ionic、RxJS、Vows/BDD 等就出现在我们的前端技术雷达屏幕里，Web Component/Polymer 甚至是我们招聘的试题。我们既是 Isomorphic（全栈同构）App 的践行者，也很可能是金融界最早最彻底大规模使用 MEAN（MongoDB、Express、Angular、Node.js）架构的团队。我们的股票交易终端证明了 HTML 5/React/Electron 技术可以成就“dead-serious”的严肃金融应用，我们的电商平台则以数百亿级的理财产品销售量，最有说服力地充当了 Angular/Node.js 在金融业的所谓“成功案例”……

时间回到 2016 年。

我们终于建立起一个新型金融科技研发组织，崇尚 Reactive 的架构风格与技术工具，时刻紧盯技术前沿，吸收大量跨界工程师，向传统 IT 注入了互联网技术基因与文化。本书的作者们，正是这个组织里有代表性的一群，他们和试验性的新技术一起成长，经受金融业对技术尝试带来的不确定性的零容忍，承担在巨额交易量下技术创新的高风险，持续学习并学以致用……这不仅需要勇气，也需要情怀，我以他们为荣。

现实中，在垂直行业具备勇气与情怀的 IT 恐怕是不多的，原因之一是因为 IT 作为企业内的“乙方”和成本中心，永远被业务线驱动而疲于奔命，无暇顾及新技术、新文化；原因二是因为以行业业务为主导的企业往往并不懂技术，也不理解技术的重要性，一位工程师选择用 JQuery 还是 Angular 开发前端，并没有人关心，而且为了“安全稳妥”起见，往往做出“保守”的选择。情怀和勇气，是垂直行业 IT 的稀罕物。

但是在一个高度同质化的行业如证券业，技术就是一个差异化竞争的决定因素，新的函数语言、新的框架、新的开发工具……差异化和技术领先，体现在这些技术细节的追求里。

本书的作者们，是这些技术细节的追求者。这一次，他们利用自己每天“正常加班”之外少得可怜的个人业余时间，凭着极大的耐心、坚持、团队协作，把其中一个细节——也就是对 Angular 的经验与理解，完全原创性地分享给读者。编写这本书，既是历时七八个月的凝聚团队的工程项目，也是年轻队员们职业生涯里一次难忘的体验，they make a difference ——首要是为了他们自己，但希望也能为你，前端技术的爱好者们！

谢谢。

梁启鸿
原董事总经理 @ 金融科技研发
广发证券信息技术部

推荐序 2

Angular's developer community in China is active and thriving. This comprehensive new book is the first originally authored book on Angular written in Chinese for a Chinese audience. The author and his team is well known in the local Angular community for his contributions. We thank him and his team for their work towards making Angular even more friendly to developers in China and hope this book will be helpful.

在中国，Angular 开发者社区非常活跃并且正在蓬勃发展。本书作者和其所在的广发证券互联网金融技术团队编写的这本新书内容全面，并且是面向中国读者的第一本中文原创书籍。本书作者所在团队所做出的贡献在当地的 Angular 社区广为人知。广发证券互联网金融技术团队的工作让 Angular 对中国开发者更加友好，我们非常感谢他们所做出的贡献，并真诚希望他们的这本书能给大家带来帮助。

Naomi Black
Technical Program Manager and Lead
Angular

推荐序 3

作为谷歌所支持和投资推广的一个跨平台的开源技术，Angular 在中国开发者中获得了越来越多的关注和应用。根据我们统计到的数据，Angular 第 1 版已经在大量的中国企业中获得了应用，其中有很多大型企业，也有小型的创业团队。

本书作者所在的广发证券互联网金融技术团队就是一个将 Angular 付诸实践的先行者。不仅如此，广发证券互联网金融技术团队还快速升级了 Angular，创造了中国企业中采纳 Angular 最早的实践先例之一。

我特别高兴看到本书作者和其所在的团队能联合撰写这本书，他们把使用 Angular 的开发经验向业界进行了分享，同时也详细介绍了 Angular 的各项特性。他们的实战经验一定能帮助更多开发者快速理解使用 Angular 进行开发的价值，看到最新版 Angular 的优势。

我相信广发证券互联网金融技术团队在本书中的介绍，能帮助读者更方便地学习 Angular。他们丰富的经验和最佳实践，能鼓舞你在你的企业中使用 Angular 的信心。

我推荐这本体现了开源和分享精神的书！

预祝你有个愉快的学习经历，并且能尽快上手 Angular！

栾跃（Bill Luan）

谷歌 开发技术推广部 大中华区主管

推荐序 4

随着网络技术的突飞猛进，Web 前端正在变得空前强大和复杂，而大型 JavaScript 项目也遍地开花。TypeScript 把在其他语言中身经百战的类型系统引入到 JavaScript，从而使得大型 JavaScript 项目开发体验脱胎换骨。经过长时间的积累，TypeScript 周边的生态环境也趋于成熟。为了紧跟 JavaScript 社区的变化，这门完全开源的语言时刻关注着社区的反馈。就在过去的一年多里，TypeScript 不断有重大版本更新，以解决社区中所看到的新需求。

而自从 2.0 版本起，完全采用了 TypeScript 开发的 Angular，又将 TypeScript 的优点发挥得淋漓尽致。自从 2014 年 Angular 团队和 TypeScript 团队开始合作以来，这个跨公司的组合就一直迸发着火花。Angular 吸取了 TypeScript 的高效和完备的工具支持，又将 TypeScript 的类型系统巧妙地应用在依赖注入等核心功能上。TypeScript 也借助于 Angular 的流行，被更多的人所熟知。这健康而紧密的合作会使 Angular 和 TypeScript 越来越有竞争力，最终让开发者受益。

本书作者广发证券互联网金融技术团队是国内 Angular 社区的先行者，他们结合自己生产实战中的经验，阐述了自己对于 TypeScript 和 Angular 的独特理解，值得一读。中文资料的缺乏一直是 TypeScript 在国内推广的一大障碍；而本书 TypeScript 篇章的专门介绍，以及其结合具体场景的诸多实例，相信会对很多开发者有所帮助。预祝此书帮助你开始一段愉快的学习旅程！

李正博
TypeScript 团队成员

前言

2016 年 9 月 15 日，Angular 横空出世。鉴于 Angular 1.x 的巨大成功，加上 Angular 自身超前而颠覆式的设计，使其市场关注度水涨船高。本书是一本帮助读者对 Angular 进行快速了解、深入熟悉并用其进行实战开发的书籍。

本书概述

本书主要分为入门篇、深入篇、实战篇和延伸篇四大部分，总共 20 个章节。

第一部分：从第 1 章到第 4 章，主要讲述整个前端发展史的演进；Angular 的发展历程、核心概念及周边工具的简单介绍；快速熟悉 Angular 官方推荐的开发语言 TypeScript；最后以一个通讯录示例介绍如何搭建开发环境并快速上手 Angular。

第二部分：从第 5 章到第 12 章，主要围绕通讯录示例深入讲解 Angular 的相关知识点，包括 Angular 的运行机理与整体架构介绍、组件与变化监测相关内容、模板与管道、指令的总体介绍、服务与响应式编程 RxJS、强大的依赖注入、灵活高可用的路由机制等，最后介绍了在项目开发中与测试相关的内容。

第三部分：从第 13 章到第 18 章，主要以实现一个问卷调查系统为目标，阐述如何使用 Angular 进行项目实战。主要内容包括项目背景介绍、开发环境的搭建、整体技术架构分析、用户管理及问卷编辑等页面的实现细节等，最后讲解了项目的构建流程及最佳实践。

第四部分：从第 19 章到第 20 章，主要讲述 Angular 的两个延伸应用，每个应用均包含完整的实战例子。其中第 19 章讲解 Angular 的混合应用开发，即 ionic；而第 20 章则讲述 Angular 的服务端渲染技术。

谁适合这本书

本书的主要目标读者是有一定 JavaScript 开发能力的新人，有 Angular 1.x 相关经验的开发者，有 Java、C# 等后端语言编程经验的人，或者想通过本书快速了解 Angular 掌握更多新鲜理念的资深工程师等。

如何阅读此书

本书基于 Angular 5.0 版本进行讲解。

本书按照由低到高的难度变化思路进行撰写。第一部分适合刚接触 Angular 的读者进行细致的阅读，如果已有相关基础或比较熟悉 Angular 的读者可以跳过第一部分，直接学习第二部分深入理解或者第三部分项目实战。

全书的插图采用统一的绘图风格，以手绘风格的形式表现出来，力求简洁，如遇部分难懂之处可配合上下文进行解读。

本书包含诸多代码段，这些代码段可分为两类，一类是比较完整独立的，跟着编写并能看到运行效果的示例代码；另一类是辅助学习的代码段，以介绍概念知识点为主，力求减少不相关代码的干扰，通常只截取最核心的片段，并以伴有省略号的形式出现。本书涉及的三个主要示例的源码也已通过 GitHub 开源，网址如下所示，感兴趣的读者可以下载运行，辅助对本书相关知识的学习理解。

- Hello World 例子：<https://github.com/angular-programming/angular-hello-world>
- 通讯录例子：<https://github.com/angular-programming/angular-contacts-demo>
- 问卷调查系统：<https://github.com/angular-programming/angular-questionnaire>
- ionic 例子：<https://github.com/angular-programming/ionic-contacts-demo>

为了加强对相关知识点的理解，本书也加入了一些旁注，对内容进行相关补充。部分较为深入但不常用的知识点，将以扩展阅读或者批注的形式展现。

勘误和支持

由于笔者水平有限，又是团体作战，且 Angular 更新迭代比较快，加上书籍撰写的时间比较仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。读者可以把书中发现的问题或建议通过在 GitHub 上提 Issue 的方式反馈给我们，网址如下所示，我们会尽快回复大家的疑问，并依据收集的信息整理修正。

<https://github.com/angular-programming/issues/issues>

读者也可登录博文视点官网 <http://www.broadview.com.cn/34272> 下载本书代码或提交勘误信息。一旦勘误信息被作者或编辑确认，即可获得博文视点奖励积分，可用于兑换电子书。读者可以随时浏览图书页面，查看已发布的勘误信息。

致谢

首先，感谢电子工业出版社的张春雨、刘佳禾等编辑及排版白涛老师，自始至终给予我们强有力的帮助和支持。如果当时没有春雨老师的邀请，我们可能就不会有写书的冲动，也就不会有本书的诞生了。

其次，要感谢广发证券互联网金融技术团队的全体小伙伴们。本书是整个团队（参与写作的人数多达 22 人）在繁忙工作之余利用琐碎的业余时间完成的，其难度不亚于一次大项目的协作，如果没有大家的紧密协作和坚持不懈，这本书也是不可能完成的，所以非常感谢以下作者的辛苦付出。

章节	作者
整体内容审校	吴炳杰、张森、高海浪、汤桂川、李仲辉、闫学凯、唐明、梁景湛
第 1 章 前端风云	汤桂川
第 2 章 Angular 简介	高海浪
第 3 章 TypeScript 入门	张森、姚云萍、郭力恒
第 4 章 快速入门	钱骞、吴炳杰
第 5 章 Angular 架构总览	李仲辉
第 6 章 组件	梁景湛、唐明
第 7 章 模板	黄晓婷、袁野
第 8 章 指令	龚麒
第 9 章 服务与 RxJS	邓玉龙、吴冠鹏
第 10 章 依赖注入	张森、姚云萍
第 11 章 路由	李远、郭伟
第 12 章 测试	李泽扬
第 13 章 问卷调查系统简介	闫学凯、王扬
第 14 章 项目起步	闫学凯
第 15 章 问卷编辑模块	闫学凯
第 16 章 我的问卷模块	闫学凯
第 17 章 用户管理模块	杨宾生

续表

章节	作者
第 18 章 项目构建和最佳实践	王扬
第 19 章 移动开发框架：ionic 介绍与实战	张淼
第 20 章 服务端渲染	梁景湛、唐明

目录

第一部分 入门篇

1	前端风云	2
1.1	故事的起点	2
1.2	AJAX 王者归来	3
1.3	工具库的流行	3
1.4	百家争鸣	3
1.5	走进前端新时代	4
1.6	小结	6
2	Angular 简介	7
2.1	历史回顾	7
2.1.1	AngularJS 1.x 起源	7
2.1.2	AngularJS 1.x 迭代之路	8
2.1.3	初生的 Angular	9
2.1.4	快速发展的 Angular	10
2.1.5	Angular 4 和后续语义版本	11
2.1.6	开发语言之选	13
2.2	Angular 简述	14
2.2.1	核心概念	14
2.2.2	平台简介	16
2.2.3	平台亮点	18

2.3	小结	19
3	TypeScript 入门.....	20
3.1	TypeScript 概述	20
3.1.1	概述	20
3.1.2	安装	21
3.2	基本类型	22
3.2.1	布尔类型.....	22
3.2.2	数字类型.....	22
3.2.3	字符串类型	23
3.2.4	数组类型.....	23
3.2.5	元组类型.....	23
3.2.6	枚举类型.....	23
3.2.7	任意值类型	24
3.2.8	null 和 undefined	24
3.2.9	void 类型	25
3.2.10	never 类型.....	26
3.3	声明和解构.....	26
3.3.1	let 声明	27
3.3.2	const 声明	28
3.3.3	解构	28
3.4	函数.....	30
3.4.1	函数定义.....	30
3.4.2	可选参数.....	30
3.4.3	默认参数.....	31
3.4.4	剩余参数.....	32
3.4.5	函数重载.....	32
3.4.6	箭头函数.....	33
3.5	类	34
3.5.1	类的例子.....	34
3.5.2	继承与多态	34
3.5.3	修饰符	35

3.5.4	参数属性.....	37
3.5.5	静态属性.....	37
3.5.6	抽象类.....	38
3.6	模块.....	39
3.6.1	概述.....	39
3.6.2	模块导出方式.....	39
3.6.3	模块导入方式.....	40
3.6.4	模块的默认导出.....	41
3.6.5	模块设计原则.....	42
3.7	接口.....	44
3.7.1	概述.....	44
3.7.2	属性类型接口.....	44
3.7.3	函数类型接口.....	45
3.7.4	可索引类型接口.....	46
3.7.5	类类型接口.....	46
3.7.6	接口扩展.....	47
3.8	装饰器.....	48
3.8.1	概述.....	48
3.8.2	方法装饰器.....	49
3.8.3	类装饰器.....	50
3.8.4	参数装饰器.....	52
3.8.5	属性装饰器.....	53
3.8.6	装饰器组合.....	53
3.9	泛型.....	55
3.10	TypeScript 周边.....	56
3.10.1	编译配置文件.....	56
3.10.2	声明文件.....	57
3.10.3	编码工具.....	58
3.10.4	展望未来.....	59
3.11	小结.....	59

4 快速入门.....	60
4.1 Hello World 例子	60
4.1.1 准备工作.....	60
4.1.2 构建项目	61
4.2 通讯录例子	66
4.2.1 背景介绍.....	66
4.2.2 架构设计.....	68
4.3 小结	74

第二部分 深入篇

5 Angular 架构总览.....	76
5.1 核心模块介绍	76
5.1.1 组件	77
5.1.2 模板	81
5.1.3 指令	83
5.1.4 服务	84
5.1.5 依赖注入.....	84
5.1.6 路由	86
5.2 应用模块	89
5.3 源码结构介绍	92
5.4 小结	93
6 组件.....	94
6.1 概述	94
6.1.1 模块化介绍	94
6.1.2 组件化标准	96
6.1.3 Angular 的组件.....	99
6.2 组件基础	100
6.2.1 创建组件的步骤.....	100
6.2.2 组件的基础构成.....	101
6.2.3 组件与模块	108

6.3	组件交互	113
6.3.1	组件的输入、输出属性	113
6.3.2	父组件向子组件传递数据	114
6.3.3	子组件向父组件传递数据	120
6.3.4	其他组件交互方式	121
6.4	组件内容嵌入	124
6.5	组件生命周期	128
6.5.1	概述	128
6.5.2	生命周期钩子	128
6.6	变化监测	130
6.6.1	数据变化的源头	131
6.6.2	变动通知机制	132
6.6.3	变化监测的响应处理	134
6.7	扩展阅读	140
6.7.1	元数据一览表	140
6.7.2	元数据说明	141
6.7.3	深入理解 Zone.js	150
6.7.4	不依赖 Zone.js 的 Angular	154
6.8	小结	155
7	模板	156
7.1	模板语法概览	156
7.2	数据绑定	158
7.2.1	概述	158
7.2.2	插值	160
7.2.3	模板表达式	160
7.2.4	属性绑定	162
7.2.5	事件绑定	165
7.2.6	双向数据绑定	168
7.2.7	输入和输出属性	169
7.3	内置指令	170
7.3.1	NgClass	170

7.3.2	NgStyle.....	170
7.3.3	NgIf	171
7.3.4	NgSwitch	172
7.3.5	NgFor	172
7.4	表单	173
7.4.1	模板表单例子	174
7.4.2	表单指令	175
7.4.3	自定义表单样式	184
7.4.4	表单校验	186
7.5	管道	189
7.5.1	管道介绍	189
7.5.2	内置管道	190
7.5.3	自定义管道	196
7.5.4	管道的变化监测	198
7.6	扩展阅读	202
7.6.1	安全导航操作符	202
7.6.2	双向绑定的原理	202
7.7	小结	204
8	指令	206
8.1	概述	206
8.1.1	指令分类	208
8.1.2	内置指令	210
8.2	自定义属性指令	219
8.2.1	实现属性指令	219
8.2.2	为指令绑定输入	221
8.2.3	响应用户操作	223
8.3	自定义结构指令	224
8.3.1	实现结构指令	225
8.3.2	模板标签与星号前缀	227
8.3.3	NgIf 指令原理	229

8.4	扩展阅读	231
8.5	小结	235
9	服务与 RxJS	237
9.1	Angular 服务	237
9.1.1	概述	237
9.1.2	使用场景	238
9.2	HTTP 服务	242
9.2.1	HttpModule	242
9.2.2	HttpClientModule	254
9.3	响应式编程	262
9.3.1	概述	262
9.3.2	ReactiveX	264
9.4	RxJS	266
9.4.1	创建 Observable 对象	266
9.4.2	使用 RxJS 处理复杂场景	266
9.4.3	RxJS 和 Promise 的对比	267
9.4.4	“冷”模式下的 Observable	268
9.4.5	RxJS 中的 Operator	269
9.4.6	Angular 中的 RxJS	273
9.5	小结	277
10	依赖注入	278
10.1	依赖注入介绍	279
10.2	Angular 依赖注入	282
10.2.1	概述	282
10.2.2	在组件中注入服务	285
10.2.3	在服务中注入服务	287
10.2.4	在模块中注入服务	288
10.2.5	层级注入	290
10.2.6	注入到派生组件	295
10.2.7	限定方式的依赖注入	297

10.3	Provider.....	300
10.3.1	概述	300
10.3.2	Provider 注册方式	302
10.4	扩展阅读	305
10.5	小结	308
11	路由.....	309
11.1	概述	309
11.2	基本用法	311
11.2.1	路由配置	311
11.2.2	创建根路由模块.....	312
11.2.3	添加 RouterOutlet 指令	312
11.3	路由策略	313
11.3.1	HashLocationStrategy 介绍.....	314
11.3.2	PathLocationStrategy 介绍	315
11.4	路由跳转	316
11.4.1	使用指令跳转	317
11.4.2	使用代码跳转	319
11.5	路由参数	321
11.5.1	Path 参数.....	321
11.5.2	Query 参数	324
11.5.3	Matrix 参数.....	326
11.6	子路由和附属 Outlet	326
11.6.1	子路由	326
11.6.2	附属 Outlet	328
11.7	路由拦截	330
11.7.1	激活拦截与反激活拦截	330
11.7.2	数据预加载拦截.....	334
11.8	模块的延迟加载	337
11.8.1	延迟加载实现.....	337
11.8.2	模块预加载	339
11.8.3	模块加载拦截.....	341

11.9 小结	342
12 测试	343
12.1 概述	343
12.2 单元测试	344
12.2.1 概述	344
12.2.2 常用测试框架	345
12.2.3 Jasmine 介绍	345
12.2.4 Karma 介绍	350
12.2.5 Karma 结合 Jasmine 测试	350
12.3 Angular 单元测试	355
12.3.1 概述	355
12.3.2 独立单元测试	358
12.3.3 测试工具集	362
12.4 端到端测试	370
12.4.1 概述	370
12.4.2 Protractor 介绍	371
12.5 小结	374
 第三部分 实战篇	
13 问卷调查系统简介	376
13.1 项目背景	376
13.2 主要特性	377
13.2.1 首页和帮助页	378
13.2.2 问卷编辑页	378
13.2.3 我的问卷页	378
13.2.4 用户管理页	379
13.3 产品设计	379
13.4 小结	380

14 项目起步	381
14.1 Angular CLI.....	381
14.1.1 简介	381
14.1.2 常用命令介绍	382
14.2 其他技术选型	391
14.2.1 UI 样式库	391
14.2.2 后端服务器	391
14.3 环境搭建	392
14.3.1 搭建前端环境	392
14.3.2 引入样式库	393
14.3.3 搭建后端环境	394
14.4 目录结构介绍	396
14.5 首页开发	397
14.6 导航栏开发	401
14.7 小结	402
15 问卷编辑模块	403
15.1 概述	403
15.1.1 特性管理模块	403
15.1.2 功能设计	406
15.1.3 数据模型	407
15.2 问卷编辑模块开发	410
15.2.1 问题选择组件	410
15.2.2 问题组件	414
15.2.3 问卷组件	425
15.2.4 问卷服务	431
15.2.5 问卷大纲组件	438
15.3 小结	441
16 我的问卷模块	442
16.1 问卷列表	443
16.1.1 问卷列表项	443

16.1.2	显示问卷列表	445
16.1.3	显示问卷详情	447
16.2	问卷操作	449
16.2.1	发布后的问卷页面	450
16.2.2	问卷操作组件	453
16.3	小结	456
17	用户管理模块	457
17.1	开发简单注册页	458
17.2	表单控件组件	460
17.2.1	定义表单控件	460
17.2.2	校验表单控件	461
17.2.3	表单安全	464
17.3	用户注册功能开发	465
17.3.1	用户注册服务	465
17.3.2	组件的逻辑	466
17.3.3	注册接口开发	469
17.4	权限管理	470
17.5	小结	473
18	项目构建和最佳实践	475
18.1	项目构建	475
18.1.1	代码质量检查	475
18.1.2	测试	476
18.1.3	打包	478
18.1.4	容器化	479
18.2	最佳实践	479
18.2.1	单一职责	480
18.2.2	命名约定	480
18.2.3	编码约定	483
18.2.4	Angular 模块约定	487
18.2.5	组件相关约定	487

18.2.6 指令相关约定	489
18.2.7 服务相关约定	490
18.2.8 其他	491
18.3 小结	492

第四部分 延伸篇

19 移动开发框架：ionic 介绍与实战 494

19.1 移动开发	494
19.1.1 背景介绍	494
19.1.2 四种开发模式	495
19.1.3 技术选型	495
19.2 ionic 平台介绍	496
19.2.1 概览	496
19.2.2 Cordova	498
19.2.3 环境搭建	499
19.2.4 组件开发	501
19.2.5 路由和导航	503
19.3 ionic Native	507
19.3.1 插件介绍	507
19.3.2 插件使用	508
19.3.3 插件开发	509
19.4 样式和主题	509
19.4.1 平台样式	509
19.4.2 主题	511
19.4.3 全局变量	512
19.4.4 工具属性	513
19.4.5 Iconfont	514
19.5 ionic CLI	515
19.6 通讯录实例	518
19.6.1 项目搭建	519
19.6.2 主页面	520

19.7 小结 525

20 服务端渲染.....527

20.1 概述 527

20.2 客户端渲染的局限性 528

20.3 服务端渲染的局限性 529

20.4 Angular Universal 介绍 531

20.5 将通讯录例子改造成 Angular Universal 的方式 533

20.6 服务端渲染的进阶实践..... 540

 20.6.1 服务端数据的同步 541

 20.6.2 使用依赖注入解决环境差异..... 544

 20.6.3 使用 Preboot 解决事件脱节 546

20.7 小结 549

第一部分

入门篇

- 前端风云
- Angular 简介
- TypeScript 入门
- 快速入门

1 前端风云

从 20 世纪 70 年代互联网出现开始，富有创造力的人们便开始了各种有趣的尝试。站在悠长的前端演进史上，从被放大的时间线上看，许多框架与工具都是那么的渺小，而缩放到它们辉煌的那些年上，却又是那么浓重的一笔。

作为本书的开篇，本章主要介绍各种前端技术背景、作用及 JavaScript 的发展历程。最后，介绍现今前端技术的理念和趋势。

1.1 故事的起点

1995 年，任职于网景公司的 Brendan Eich 为 Netscape Navigator 2.0 创造出了 JavaScript（简称 JS，最初称为 LiveScript）。在最初的一段时间里，受限于当时网站的开发模式，JavaScript 能发挥价值的地方非常有限，一直被业界当成小玩具般的编程语言。后来随着微软 IE 3.0 与 JScript 的出现，几种不同版本的浏览器端脚本语言同时存在，这就促使了后续 JavaScript 语言标准化的建立。

1997 年 6 月，ECMA 的第 39 号技术专家委员会（Technical Committee 39，简称 TC39）制定了第 1 版的 ECMA-262 标准，它定义了 ECMAScript（简称 ES）这门全新的脚本语言。接着在 1999 年，带着多种新特性的 ECMAScript 3 与 IE 5 正式发布，同年年末 HTML 4.01 也闪耀登场。在种种的新机遇下，前端领域逐渐发展起来。

1.2 AJAX 王者归来

2004 年 4 月 1 日，Google 公司发布了 Gmail，这是 Paul Buchheit 主导的团队花费近三年时间开发的大型网页应用，在“愚人节”这一天带给了 Web 世界意义非凡的变化。而在 20 世纪 90 年代中期设计的 Hotmail 和雅虎电邮，都使用了原始的 HTML 语言来编写界面，服务器处理每一次请求都需要重新加载网页，这使得响应速度与用户体验都非常糟糕，特别是在网速缓慢的年代里。而在 Gmail 中，使用了与服务器高度互动的 JavaScript 脚本，实现了更好的局部刷新效果，让交互体验更接近常规软件。后来在 2005 年的一篇文章中将这些技术命名为 AJAX（Asynchronous JavaScript And XML），它也成了现在 Web 开发的标准做法。

AJAX 的出现不仅仅是提升了加载速度那么简单，而是让前端可以承担更多的工作，为前后端开发的解耦创造了可能。虽然 AJAX 是一些旧有技术的集合，但却以一种新的姿态，披着荣光归来。随着 AJAX 的崛起与 Web 2.0 的出现，前端开发史终于迎来了新一轮革新高潮。人们越来越关注到前端这一领域，并开始纷纷为之添砖加瓦。

1.3 工具库的流行

在这高速发展的时期，前端项目变得越来越复杂，把在这个阶段遇到的一些前端开发问题如浏览器兼容、操作 DOM 的复杂度等逐渐放大。于是社区内部也涌现出一批如 Dojo、Prototype、MooTools、jQuery 等代码库来对其进行各种补充修正，而其中以 John Resig 在 2006 年发布的 jQuery 尤为出彩。jQuery 以其巧妙的接口封装、简洁的链式写法和高效的选择器实现，再加上丰富的插件体系，不需要关注不同浏览器的接口差异问题，大大提升了前端开发的生产力。在一段时间里很多前端工程师的招聘要求中都提及需要熟悉甚至精通 jQuery，可见其使用的广泛性与重要性。

伴随着各种 DOM 操作库与模板引擎的出现，再加上相应的 UI 组件库的普及，前端社区内也出现了各类前端架构化的尝试与小范围的实践。不少公司的项目也由原先后端主导的模式转向富前端化，将更多的逻辑交由前端来实现，而后端仅提供更为底层的数据处理与部署运维。

1.4 百家争鸣

随着前端生态社区日渐蓬勃发展，大型前端架构的深入实践与工程化等新问题被不断提出。而在这风起云涌的大时代背景下，旧有的 jQuery 时代技术应对这些新问题时时早已有心无力。彼时，人们为了追求更快的页面访问体验，提出了单页 Web 应用

(Single Page Application, 简称 SPA) 的概念, 前端社区中各类架构概念的迁移与实践也不断出现。

1979 年提出的 MVC (Model-View-Controller) 架构在各类编程语言中都有相应实现, 在 JavaScript 中也不例外。在 2010 年左右, 一批实现了 MVC 架构的优秀框架如 Backbone、Batman、CanJS 等不断涌现。其中以 Jeremy Ashkenas 创建的 Backbone 最为出名, 一度火遍大江南北, 可以称之为实现 SPA 的利器。

而在 2005 年, 微软提出了一种新的架构模式——MVVM (Model-View-ViewModel), 其最主要的特点是双向绑定技术, 解决了 Model 层和 View 层的强耦合问题。在 JavaScript 中也有一批实现了这种架构的框架, 如 AngularJS、Knockout、Ember、Vue 等。在这众多框架中以诞生于 2009 年的 AngularJS 尤为出名, 其自身定位为 MVW (Model-View-Whatever) 模式, 并以双向数据绑定技术、简洁易用的模板语法、强大的依赖注入功能吸引了众多拥护者。各大框架之间长年累月的角逐, 也在用户的选择中逐渐分出了高低。在 AngularJS 正式发布后的一段时间里, 它无疑就是当时的王者, 这一用户量最大的前端框架, 也成为了实现 SPA 的最佳选择。

AngularJS 诞生的 2009 年, 在前端史上无疑是意义非凡的一年。

在这一年里, ECMAScript 5 发布, 这宣告了 ECMAScript 4 的最终“流产”。ECMAScript 4 的“流产”事件却也为后续的 ECMAScript 6 的定案做了铺垫。同年, 技术社区内发布了一项名为 CommonJS 的规范, 解决了 JavaScript 模块化的问题, 而后衍生出了 AMD (代表库 RequireJS)、CMD (代表库 Sea.js) 等模块化规范, 使得前端模块化越来越流行。也在同一年, Ryan Dahl 基于 CommonJS 规范和 Google V8 引擎创造出了 Node.js。它是服务端 JavaScript 的运行环境, 可以让前端工程师更简单地进行后端开发。Node.js 的出现, 也让大量前端自动化工具如 Grunt、Gulp、Webpack 等陆续被创造出来。同年年底, CoffeeScript 这一 JavaScript 的转译语言出现了, 它只保留了 JavaScript 语言的精髓, 提供了大量的语法糖, 跟进了 ECMAScript 的部分新标准特性, 让前端代码书写变得更为简洁, 在很大程度上提升了开发效率。与之类似的, 还有在 2012 年发布的 TypeScript, 身为 JavaScript 的超集, 更是把强类型特性带到了前端社区, 通过类型检查更容易发现问题。TypeScript 紧跟着 ECMAScript 标准的实现, 吸引了大量粉丝, 火爆度逐日递增。

1.5 走进前端新时代

2014 年 9 月, HTML 5 发布, 标志着大量功能强大的接口特性被确定下来。而在发布之前, 其中的很多特性已被大量使用与实现。在这样的大背景下, 激进而又爱好捣鼓新玩意的前端界也涌现出大量实用且具颠覆性的新东西: 使用 ionic、React Native、

NativeScript 等技术，前端工程师可以进行移动（iOS、Android）App 开发；使用 NW.js、Electron，前端工程师可以进行桌面（Windows、Mac、Linux）应用开发；使用 Node.js 技术，前端工程师可以进行同构（Universal）服务器端应用开发。由此前端工程师的工作能力范畴被放大，市场价值增加了很多。

正当 MVVM 框架大行其道之时，2013 年 Facebook 推出了 React.js 这一富有市场竞争力的杀手锏式项目。不同于 MVVM 框架，React.js 只是一个视图层的 JavaScript 库，其数据更新机制来源于游戏开发领域的理念，采用了“整体全局刷新”的模式，但由于使用了自身的 Virtual DOM 技术，从而避免了昂贵的 DOM 操作开销，加上其高效的 DOM Diff 算法能精准地对发生变化的节点进行局部更新，使得整体性能非常优秀。

在构建网页程序的架构设计上，Facebook 提出了 Flux 的概念。不同于 MVC 架构，Flux 是一个包含了 Action、Dispatcher、Store 和 View（React 组件）的单向数据流的架构（模式）。市场上有很多实现 Flux 架构的库，其中以 React Hot Reload 的作者写的 Redux 尤为出名，Redux 将自身定位为一个可以预测的状态容器，并且大量采用了新标准的语法，提升了开发效率。

React 社区吸收了很多函数式编程的理念，而函数式编程的一大特点是尽可能减少可变动的部分。过去有人提出 Om（用 ClojureScript 写的 React）在速度上比原生 JavaScript 版本的 React 快了非常多，震惊了整个 React 社区。究其背后，其中一个很重要的原因是 ClojureScript 这门函数式编程语言使用了不可变数据类型（Immutable Data）。它的优点在于节省了内存，并降低了可变数据带来的复杂度等。受此启发，React 社区也冒出了 Immutable.js，优化了 JavaScript 这门弱类型语言对引用对象的变化检测性能问题，更好地提升了性能。

伴随着前端社区快速的迭代发展，在 2015 年 6 月 17 日，ECMAScript 6 正式发布。此新标准从开始制定到最终发布历时 15 年，带来了大量新特性与语法糖，以及众多已有特性的强有力扩展。这是继 ECMAScript 3 之后，JavaScript 语言的又一次意义重大的革新，意味着前端界即将迎来一个崭新的时代。同年 TC39 决定以后每年都将当前已经标准化的特性发布出来，并以年份进行版本命名，所以 ECMAScript 6 也被称为 ECMAScript 2015。在新标准未发布之前，前端社区内早已流行起一些 ECMAScript 标准的编译转换工具如 Babel（起初称为 6to5）和 Google 的 Traceur 等，这让开发者可直接在项目中使用未来的标准语法来编写代码，最终再通过这些编译工具转换为 ES 5、ES 3 的代码。其中作为后起之秀的 Babel 是由当时还是高中生的 Sebastian McKenzie 在 2014 年 9 月着手开发的，如今它以更全的兼容性、更强的生态圈及丰富的插件体系显得更为耀眼，并成为了当前的主流。

在其他前端框架社区高速发展与新标准诞生的时期，AngularJS 正背负着臃肿的架构、过时的概念和低效的性能问题等沉重的历史包袱，而在时间线的另一头，又有新的故事发生——Angular 正缓缓地走入我们的视线。

1.6 小结

通过本章的学习，读者了解到了 JavaScript 与其标准 ECMAScript 的发展历史，也了解到了前端社区发展历程的一些标志性事件及其影响。随着前端领域新技术的流行，学习成本变得越来越高，这同时也意味着能做的事情越来越多了。新的机遇与挑战并存，只有扬帆起航，战胜困难，才能抵达胜利的彼岸。

第 2 章将深入了解 Angular 的历史与架构、周边工具等内容，让我们开始这一趟 Angular 之旅吧。

Angular 简介

通过第 1 章的介绍，我们对近几年 Web 开发现状和趋势有了较为完整的认识。同时也了解了很多概念，如 MVC、MVVM、Flux 和 Immutable Data 等，这些概念也慢慢地融入到各类项目的开发实践中。本章将会正式讲解 Angular，看看 Angular 受到了哪些思维的影响，以及 Angular 本身如何引领了一代开发风潮。

本章内容将首先回顾 Angular 的历史，了解它是怎么从个人项目 GetAngular 转型为 Google 内部流行的官方项目的；然后介绍 AngularJS 1.x 各个版本迭代引入的各种特性；最后讲述从 Angular 诞生到发布正式版的历程。此外，也会对 Angular 的主要特点和概念进行简单讲解，便于读者在后续章节的详细论述前有个直观的全局认识。

2.1 历史回顾

2.1.1 AngularJS 1.x 起源

2009 年，Misko Hevery 和 Adam Abrons 在业余时间创造了 AngularJS 1.x。起初，它叫作 GetAngular，是 Web 设计师和前后端工程师用于沟通的端到端设计开发工具。

随后，Misko Hevery 在 Google 接手了 Feedback 的开发，经过 6 个月将近 1.7 万行代码量的功能迭代，代码库越来越大，开发和维护变得举步维艰。Misko 找到了他的经理 Brad Green，打赌花两周的时间用 GetAngular 重写该项目。最后 Misko 仅花了三周时间，

并且代码行数从原来的 1.7 万行精简到 1500 行。Brad 十分看好 GetAngular 项目，于是把它改名为 AngularJS，并在 Google 内组建了团队专职进行开发和维护。

后来适逢 DoubleClick 被 Google 收购，AngularJS 被用来重写它的部分业务逻辑，其效率之高令人称奇。至此，AngularJS 在公司内部一鸣惊人。Google 管理层觉得应该将更多的人力和资源投入到 AngularJS 团队，让他们集中精力开发 AngularJS 框架和周边工具，并且在公司内外做产品开发，大力地进行运营和推广。

2.1.2 AngularJS 1.x 迭代之路

在进入开发快车道后，AngularJS 1.x 的版本迭代有条不紊地进行着。

2012 年 3 月，AngularJS 进入 1.0.0 发行候选版开发，前前后后 12 个 rc 版本历时 3 个月，才在 6 月中旬发布了 1.0.0 版本（temporal-domination）。这个正式版本集成了 Angular 应用习以为常的诸多概念：

- 把 MVVM 这种高效开发界面的技术变得流行起来。
- 引入 Directive 指令来扩展 HTML 的语义。
- 引入 Controller 和 Scope 来构建应用逻辑。
- 引入 Service、Factory 和 Provider 等来为公共和数据逻辑提供抽象。
- 基于 ngModel 双向数据绑定的表单格式化和验证。

在 1.0 版本发布后，Angular 核心团队在接下来的几个月内采用了 1.1 版本和 1.2 版本同时开发的版本迭代方式，其中 1.1 版本开发新功能特性，引入不兼容的改造，待稳定运行发版后合入 1.2 版本。

2013 年 11 月，整合了 1.1 版本功能的 1.2 版本（timely-delivery）姗姗来迟，它带来的功能和调整如下：

- 把 ngRoute 路由抽取到单独的模块中。
- 引入 ControllerAs 语法，使得模板中的数据绑定更加清晰（而不是分不清层级的 Scope）。
- ngRepeat 支持 track by 功能。
- \$http 等服务遵从 Promises/A+ 标准。

同时，Angular 团队花了大量时间去优化、简化并提升了文档质量和官方网站的浏览体验。

2014 年 10 月中旬，在 1.3 版本（superluminal-nudge）中，AngularJS 1.x 不再继续支持 IE 8。同时提供了如下新功能：

- 单次绑定，在模板中使用“::”的表达式。
- ngMessages，利用该模块来更好地做表单检验消息提示。
- ng-model-options 对 ngModel 的更细粒度的控制。



在同年的 4 月，微软宣布停止对 Windows XP 系统的支持，这也意味着微软不再支持主要运行于该系统上的 IE 8 浏览器。

2015 年 5 月底，Angular 团队终于发布了 1.4 版本（jaracimrman-existence）。该版本中有超过 400 条提交，同时优化了文档，修改了 100 多个 bug，以及新增了 30 多个新特性，其中包括：

- 重写动画模块，修复大量的遗留 bug。
- Router 模块，为 AngularJS 1.x 和 Angular 提供了强大且一致的路由方案。

2016 年 2 月，Angular 团队又发布了 AngularJS 1.5 版本，该版本的主题就是要和 Angular 做进一步整合，提供更接近于 Angular 应用的书写体验，如组件式的开发、定义组件指令、生命周期钩子，等等。

2.1.3 初生的 Angular

2014 年 3 月，官方博客就有提及在为新的 Angular 做设计和开发，所有的设计文档都公布在 Google 云盘中。博客宣称该框架有如下特点：优先为移动应用设计、更快速的变化监测、ES 6 原生模块化引入、采用状态、支持整合认证授权和缓存视图的新路由、有持久化支持离线使用的存储层，等等，大家为之兴奋不已！

在 2014 年 9 月下旬召开的 NG-Europe 大会上，Angular 首次亮相，与公众见面。它的接口和概念变化在很多 AngularJS 1.x 开发者中引起了不小的争议。这些变化包括：

- 引入 Component，统一控制器和模板，向 Web Components 的标准看齐。
- 引入 AtScript，对 TypeScript 和 ES 6 语法增强，添加可选运行时类型和注解，来帮助大型团队开发复杂的应用。
- 移除 Scope 概念。



目前 Angular 已经放弃 AtScript，采用了 TypeScript 作为 Angular 官方开发语言。

这些大的改变抛弃了 AngularJS 1.x 这几年来的一些历史包袱，让经验老到的开发团队能够重新设计，结合 AngularJS 1.x 的经验教训和从外界引入的思潮（例如，参考 React 的 Virtual DOM 方案分离出的渲染来获得性能提升和平台扩展性，向 Web Components 的标准看齐，等等）。不过，这些激进的转变让 AngularJS 1.x 的开发者感到不习惯，很多人开玩笑说：“现在用 AngularJS 1.x 这个注定很快要被淘汰的框架开发业务代码……”。但从长远来看，尽管当时遭受各种抨击，但这个破釜沉舟的决定还是非常正确的，它成就了现在具有高性能、高开发效率、丰富的扩展能力特点的 Angular。

2015 年 4 月 30 日，Angular 团队宣布将 Angular 从最初的 alpha 版本转到开发者预览版本。

同时官方先后引入了 ngUpgrade 和 ngForward（官方已经不再维护），支持向现有的 AngularJS 1.x 应用中集成 Angular 的代码，为那些从 AngularJS 1.x 向 Angular 迁移的应用提供了解决方案。

2.1.4 快速发展的 Angular

通过 alpha 版本和开发者预览版本，Angular 团队和 Google 内部多个大项目组的同事紧密配合，包括 AdWords 广告团队、GreenTea 内部客户关系管理软件的团队和 Google 光纤团队，这也算是在真实项目需求中检验着 Angular。事实上，2015 年年底上线的 Google 光纤产品，正是基于 Angular 框架实现的。

2015 年 12 月，Angular 开始进入 beta 版本。多个外部项目开始整合以适应新的 Angular，如：ionic 框架推出了 ionic2 计划，Telerik 加紧对 NativeScript 的整合，Rangle.io 着手开发 Batarangle 开发者工具，等等。同时 Angular.io 官网正式上线，添加了入门教程、开发者指南、参考手册等使用文档，开发者可以跟着文档一步步体验并使用 Angular。

2016 年 5 月，Angular 发布了 rc1 版本，正式进入发行候选阶段。在 6 月中旬发布了 rc2 版本，它的动画模块从底层支持多平台，合入了超过 100 项社区贡献的代码。一周后，rc3 版本发布，把新路由项目合入主代码库中（但路由模块仍然会保持独立的版本和发布周期）。在 7 月初发布的 rc4 版本中，官方对暴露出来的公共接口进行了清理，并且大幅提升了测试的灵活性和易用性。在 8 月发布的 rc5 版本中，又引入了不少新特性，如：路由支持懒加载、组件和服务支持 Ahead-of-Time (AoT) 编译、新的 NgModules 封

装方式等。在随后的 9 月，官方先后发布了 rc6 和 rc7 版本，其中 rc6 为表单引入了更多的功能（如 validator 指令绑定等），同时增强了国际化支持；而 rc7 则主要集中在问题的修复上。

通过这七次 rc 版本的迭代，Angular 已经基本趋于稳定：

- 通过广泛的使用场景验证，为外部提供的 API 接口不会再有变更。
- 框架本身被反复优化，已经能提供更好的开发体验、更小的加载包，以及更快的性能，等等。

2016 年 9 月中旬，在 Google HQ 的见面会上，官方正式发布了 Angular 版本。



官方团队会继续在 Angular Material 2、Angular Universal 等周边项目上发力，同时提供更好用的动画模块，以及小问题修复。

2.1.5 Angular 4 和后续语义版本

Angular 正式发布后，先后经过几个月的小版本升级，包括 2.1、2.2、2.3 等。

2016 年 10 月，Angular 2.1 发布并引入：

- 路由支持预加载定义为懒加载的模块。
- Angular 中的动画功能增强，引入:enter 和:leave 别名，方便使用。

在接下来的 11 月，Angular 2.2 发布并引入：

- 使用 @angular/upgrade 时 AoT 会编译 Angular 的组件和模块。
- 为方便从 1.x 到 2.x 的应用版本迁移，优化增强 Router 模块。
- 在大量使用表单的时候 AoT 编译产生的代码量有所减少。
- 添加 Angular 搭配使用 ES 5 和 ES 6/7 的文档指南。

在随之而来的 12 月，Angular 2.3 发布并引入：

- 发布第一版的 Angular 语言服务（和 IDE 编辑器整合提供模板的错误检查和类型补全功能等）。
- 组件可以利用 JavaScript 对象继承来扩展功能。
- 最新版的 zone.js，包括增强优化的调用栈信息。

大家可以看到 Angular 团队现在保持着紧凑的迭代节奏：

- 每周补丁级别的更新发布，通常只修复问题，不加入新功能。
- 每次大版本更新发布后，会有 3 个月左右的小版本发布（次要版本），主要加入一些变更不大的新功能。
- 每 6 个月会有方便迁移但有破坏性变更的大版本发布。

采用了 SEMVER 语义版本的方式来应对变更。不像之前 AngularJS 1.x 到 Angular 2，是一次完全的重写，包括全部改变的 API 和全新模式。后续从版本 2 升级到版本 5、版本 6 等不会像之前有那么大的改变——仅仅对某些核心库修改就会要求主版本更新，Angular 团队会使用工具来让升级更平滑，即使对那些破坏性的改变也能做到自动升级（通过 CLI 工具来改写对应代码等）。所以，以后我们更适合叫它 Angular，而无须加上版本后缀了（Just Angular）。



语义化版本的主要作用是让每一个版本号的添加都有其意义。这可以让开发人员迅速明白此次升级的变动情况，而且可以让第三方工具，比如 NPM 可以更便捷、更安全地进行操作。关于 SEMVER 语义化版本的详细介绍可以参阅 <http://semver.org/lang/zh-CN/>。

2017 年 3 月，跳过了 Angular 3，我们迎来了 Angular 4 版本。为什么没有 Angular 3 版本呢？主要是因为目前 Angular 框架核心代码都在统一的 GitHub 代码仓库中（github.com/angular/angular），但其中包含诸多模块需要统一版本号管理，然后作为不同的包在 NPM 上发布。这样可以避免后续维护的不便，但是由于之前的 angular router 版本号不对齐（早在 Angular 2.x 的时候就已经占用了 router v3 版本），所以核心团队决定直接升级到 Angular 4。

Angular 4 这个版本速度更快、体积更小，且向后兼容了 Angular 2.x 系列。它的更新主要集中在如下几个部分：

- 独立的动画模块，开发者可自行导入使用。
- AoT 产生的代码量减少，最高可达 60%。
- 优化了 ngif/ngfor、if/else、local variable。
- Angular Universal 从社区驱动到官方团队支持。
- 兼容 TypeScript 2.1 到 2.2，优化了模板的 source map 等功能。

同时，Angular 4 成为第一个长期支持版本，这意味着后续的新功能和补丁会得到更新和修复的保障。

未来 Angular 团队还将持续把 Angular 打造成更快、更小的开发框架，同时给 @angular/service-worker 等模块增加特性，新增 @angular/common/http 模块替换原来的 @angular/http，并且把 @angular-language-services 模块从实验性阶段逐步升级为可用阶段。

Angular 5 又是一个主版本更新，它再次体现了官方团队把 Angular 做得更小、更快、更好用的一贯目标。相比于 Angular 4，它的更新主要有：

- 构建优化器，更好地利用“tree shaking”的效果。
- 新增 Angular Universal 状态转交 API，加强服务端和客户端的数据传输能力。
- 编译器改进，增加开发环境 AoT 支持、可选保留空白符等特性。
- 新增国际化的数值、日期和货币管道。
- 新增 StaticInjector 代替 ReflectiveInjector，逐步减少依赖 polyfill。
- 提升 Zone 的速度。
- 新的路由器生成周期事件。

2.1.6 开发语言之选

优秀框架的开发离不开强大的编程语言，尤其是像 Angular 这样的大型框架。在 Angular 框架开发之路上，使用过 Dart、AtScript、TypeScript 这三种与 JavaScript 相关的开发语言，它们都在一定程度上弥补了 JavaScript 的不足，提供了更多的语法糖和新的功能特性。在严谨的工程实现上，对语言的特性和功能的要求都是苛刻的。

Dart 是 Google 寄予厚望的用于替代 JavaScript 而开发的编程语言。在 Google 内部，它被用于重写 AngularJS 1.x 来试用检验这门语言的适用性。这个重写项目叫作 Angular-Dart，它是 Hevery 在 2014 年一直集中精力实施的项目。非常有意思的是，据 Google 团队所说，AngularDart 项目的效果出乎意料的好，因为这让核心团队能够接触和产生很多新的想法，比如最早借鉴 Dart 而引入的 Zones 等特性。在 Angular 项目中，核心团队从最早的编译到 Dart 方案，改为目前独立的 Dart 版代码仓库形式的方案（因为不满意之前的编译状态和 bug 修复速度）。

AtScript 基于 JavaScript，并且扩展了微软的 TypeScript，也是一门可以最终转译成 JavaScript 的脚本语言。最早在 2014 年的 NG-Europe 大会上，Angular 团队核心开发人员宣布它为后续 Angular 的主要构建语言。起初 AtScript 被设计运行在 TypeScript 之上，同时从 Dart 引入一些有用的新特性。

不过，在 2015 年 3 月的盐湖城会议上，微软 TypeScript 和谷歌 Angular 开发团队一起宣布会把 AtScript 中的不少新功能特性在 TypeScript 1.5 版本中发布，同时 Angular 将放弃 AtScript，而使用 TypeScript 作为首选的开发语言。这是影响业界的大事，强大框架和语言的珠联璧合，给 Web 前端开发带来了新的可能。

2.2 Angular 简述

在本节会进一步介绍 Angular，从七大核心概念看其背后的设计亮点，通过分析 Angular 从框架到平台演进的过程来观察其发展趋势。

2.2.1 核心概念

Angular 框架有七大核心概念，它们是 Angular 的重要组成部分，如图 2-1 所示。

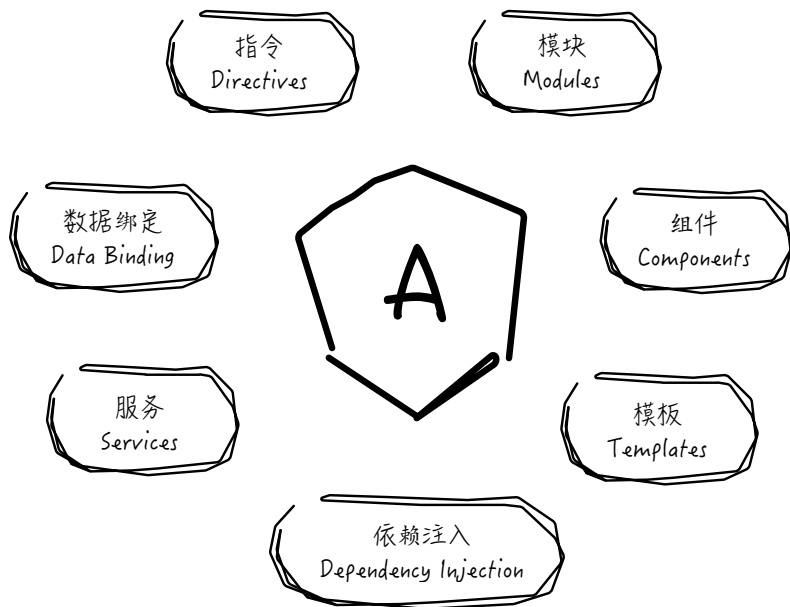


图 2-1 Angular 的七大核心概念

模块

在 Web 开发中，通过依赖全局状态或变量和保证 JavaScript 文件引入顺序来正确加载相应的类库。比如：\$ 代表 jQuery，在引入 \$.superAwesomeDatePicker 类库来实现日期选择控件前，需要确保 jQuery 已经正常载入。随着项目中的程序越来越大、文件切分越

来越细，就会需要一个成熟的模块系统（如之前介绍的 AMD、CommonJS 等）来帮助管理项目文件的依赖关系。在新的语言标准 ES 6 中，提供了 `import` 来导入在其他文件中定义的模块，且用 `export` 将诸如 jQuery 或 moment 这样的依赖导出到业务代码模块中。在后续的 TypeScript 章节中会对这些语法进行更详细的讲解。

指令与组件

在 Angular 中，指令是一个极其重要的概念。指令可以为特定 DOM 元素添加新的行为特征，从而扩展元素的功能。指令与 HTML 元素属性的使用方式非常相似，但指令的可自定义特性在一定程度上弥补了 HTML 元素属性功能的不足，这也为多样的 Web 前端开发创造了更多的可能性。

实际上，组件是指令的一种类型。以组件为基础的架构模式是现在 Web 前端开发的主流方式。不仅仅在 Angular 中，在类似的 React、Ember 或 Polymer 等框架中也是很常见的。这种开发方式就是构建一个个小的组织代码单元，每个代码单元的职责定义清晰，并且可以在多个应用中复用。例如：想使用 Google 地图组件，就在页面引入 `<google-map pointer="46.471089,11.332816"></google-map>` 这样语义化的标签。

Angular 全面支持这样的开发方式，在 Angular 中组件是“一等公民”。伴随组件而来的是组件树的概念。一般来说，每个 Angular 应用都有一棵组件树，由应用组件或者叫顶层的根组件和许多子组件及兄弟组件组成。组件树是很重要的概念，后续章节还会继续讲解。它有很多作用，比如形象地勾勒出 UI 界面的组成，这种树形结构也体现了从一个组件到另一个组件的数据流动，Angular 也依赖组件树做出合适的变化监测策略。一个博客模块的组件树例子如图 2-2 所示。

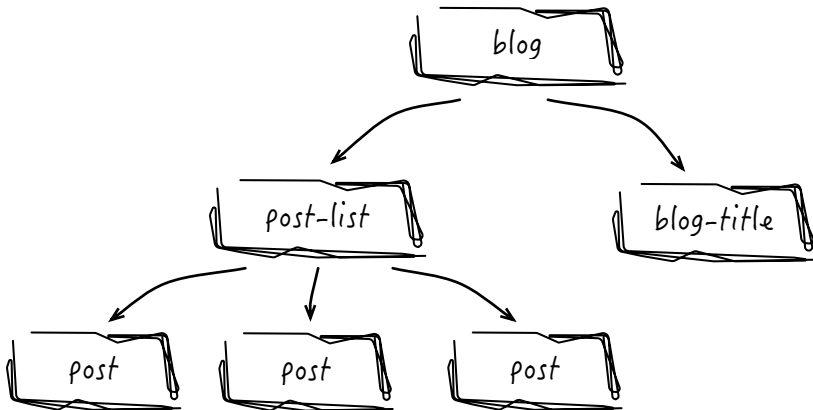


图 2-2 一个博客模块的组件树例子



变化监测是 Angular 在应用的数据变化后，用于决定哪个组件需要随之刷新的机制。

模板和数据绑定

当使用组件标签时，可以通过 `template` 或 `templateUrl` 属性引入 HTML 来描述让 Angular 渲染显示的界面内容。另外，需要数据绑定机制来实现把数据映射到模板上，或者从模板（如 `input` 控件）中取回数据。

服务和依赖注入

在 Angular 中，如果说组件是用于处理界面和交互相关的，那么服务就是开发者用于书写和放置可重用的公共功能（如日志处理、权限管理等）和复杂的业务逻辑的地方。服务可以被共享，从而被多个组件复用。在 Angular 中，一个服务就是一个简单的类。通常在组件中引用服务来处理数据和实现逻辑。

依赖注入可以帮助应用解耦，一般通过对实现服务的类加上 `@Injectable` 装饰器，同时把它注册到 `Provider`（可以在模块、其他服务、根组件或需要注入服务的上层组件中实施），从而将服务提供给调用者使用。关于依赖注入的详细讲解可以参阅第 10 章内容。

2.2.2 平台简介

Angular 的项目经理 Brad 说过，Angular 现在更像是一个平台，而不是简单的类库或者单一的框架。Angular 在技术架构上倾向于平台化设计，其跨平台开发特性使得周边生态圈变得更加繁荣兴旺，如图 2-3 所示。

Angular 框架核心包含了以下内容：

- 依赖注入
- 装饰器支持
- Zones
- 编译服务
- 变化监测
- 渲染引擎

其中, Zones 可以独立于 Angular 使用在其他地方, 并且已经提交给 TC39, TC39 也考虑将其纳入 ECMA 标准中。而渲染引擎也是平台独立的, 从而可以方便地实施在桌面软件和原生的移动客户端中。

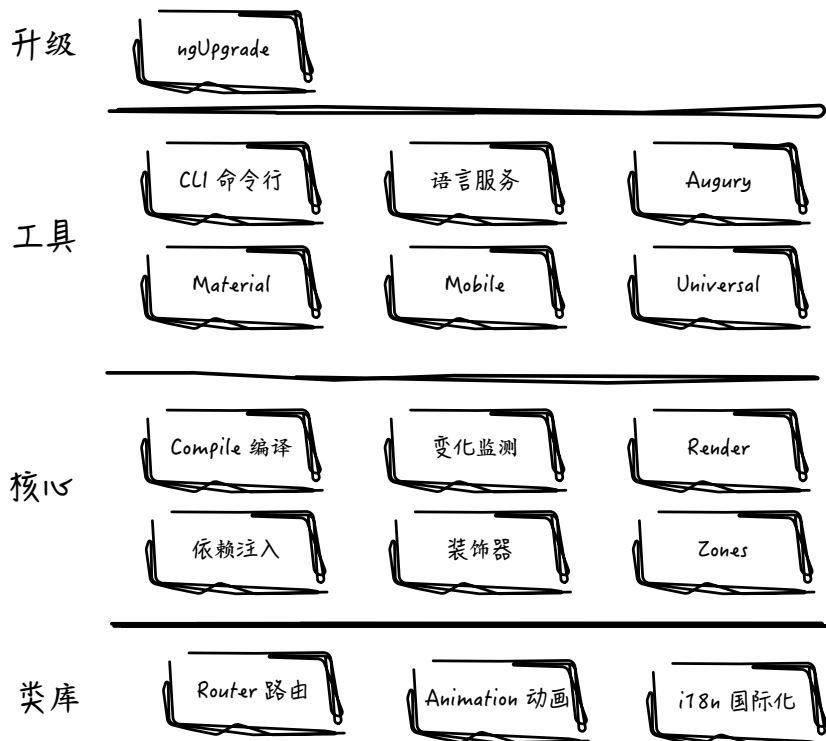


图 2-3 Angular 平台一览

在此之上, 还有不少其他的外部工具库, 类似于:

- Angular Material, Google 官方的 Material 设计风格的 UI 组件库。
- Angular Mobile Toolkit, 它提供工具和代码技巧来协助开发高性能的移动应用。
- Angular Universal, 用它实现后端渲染, 从而加速首屏渲染和实现搜索引擎优化等。

除了这些, Angular 周边也有完善的工具体系:

- Angular CLI 为开发者提供了工作流自动化解决方案。其功能涵盖了创建项目、生成组件、配置路由、代码格式化、启动开发服务器、构建测试、运行测试、预处理 CSS 样式和部署前的构建, 等等。

- 语言服务采用 TypeScript 构建，支持 IDE 中的代码补全、语法检查报错、定义跳转和方法提示等功能，从而显著提升了开发效率和编译运行前的错误发现。
- Augury（之前叫 Batarangle）用于调试、分析性能和可视化查看应用组件树，是帮助开发者快速定位问题和调优的工具利器。

当然，为了开发强大的应用，Angular 在功能开发上也提供了不少辅助模块，例如：

- i18n 模块，用于语言国际化、符号时间等本地化。
- 路由模块，用于构建多界面状态的单页应用。
- 动画模块，提供了基于声明式的书写体验和完善 Hook 节点的功能。
- Upgrade 模块，Angular 和 AngularJS 1.x 不是孤立的，通过 Upgrade 模块（原 ngUpgrade）能够方便地将使用 1.x 开发的应用升级到 2.0 以上，面向未来编码。

2.2.3 平台亮点

以上内容先后介绍了 Angular 核心概念和 Angular 平台提供的各种各样的功能，那么 Angular 相对于其他前端技术有什么特点呢？

它拥有超快的性能：

- 优化渲染速度，更快地检测变化，内部拥有性能基准的测试框架。
- 对视图进行缓存，从而实现列表流畅滚动和页面切换如丝般顺滑。
- 首屏加载更快，使用服务端渲染和小型启动库使网络加载更快。
- 移动端响应大幅度提升，原生支持各种手势、触摸等。

其中，Angular 服务端渲染（Server-Side-Render）会在后续章节中进行详细讲解和实践介绍。

它支持完善流畅的开发体验。除上面提到的 CLI 工程化的命令行工具、Augury 审查工具和 TypeScript 语言服务外，也包括：

- 官方支持的代码风格指南和检查（Lint / Style 工具）。
- 可以快速搭建项目的 Yeoman Generator、Webpack Starter 等脚手架。
- 对不同技术背景的开发者提供如 Dart、ES 5 等其他语言版本的选择。

Angular CLI 工程化流程如图 2-4 所示。

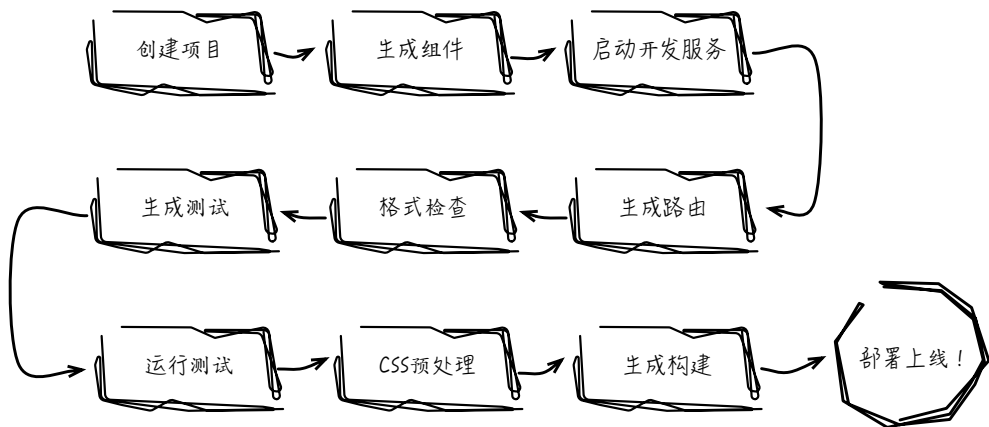


图 2-4 Angular CLI 工程化流程

它的社区和周边也强大多样。除了上面提到的 Material Design UI、Mobile Toolkit，还包括：

- Kendo UI、Onsen UI 2 等 UI 库，提供了多样化的界面方案选择。
- ionic2、NativeScript、React Native 等移动端技术，用来开发跨平台的混合或原生应用。
- Meteor 等框架，可以用来实现 JavaScript 全栈式开发和高效整合。

不得不说，基于最新 Angular 的 ionic 变得越发强大，是用 JavaScript 技术开发移动应用的不错选择。同时，利用最新的 PWA（Progressive Web App）Web 技术，能够帮助我们很好地打造移动版网站。因此，在本书后面会用专门的章节来讲解这两个热门话题。这就是你应该立即使用 Angular 的原因！

2.3 小结

通过本章的学习，相信读者对 Angular 已经有了较为直观的认识，清楚了它的历史发展轨迹，也了解了新生的 Angular 框架逐步成长为颇具潜力的大平台的发展历程。在实际项目中，我们可以使用 Angular 提供的模块、组件、模板数据绑定、服务、依赖注入和注解等特性来实施应用开发，Angular 社区也提供了各种辅助周边、功能模块和开发工具等。这最终形成了性能强劲、开发体验完善和社区周边强大的 Angular。

在第 3 章中，我们会学习 TypeScript 语言，它是构建 Angular 框架的基石语言，也是官方推荐的开发语言。

3 TypeScript 入门

通过第 2 章的介绍，相信读者已经对 Angular 的历史有了全局的认识。Angular 最终选择 TypeScript 作为其官方最主要的构建语言，对开发者来说，这意味着掌握 TypeScript 语言将更有利于高效地开发 Angular 应用。本章将结合 Angular 的使用场景来介绍 TypeScript 这门语言，相信读完本章后读者能对 TypeScript 有更深入的理解。

3.1 TypeScript 概述

3.1.1 概述

TypeScript 是由 C# 语言之父 Anders Hejlsberg 主导开发的一门编程语言。TypeScript 本质上是向 JavaScript 语言添加了可选的静态类型和基于类的面向对象编程，同时也支持诸如接口、命名空间、装饰器等特性，它相当于 JavaScript 的超集。关于 ES 5、ES 6、TypeScript 的关系如图 3-1 所示。

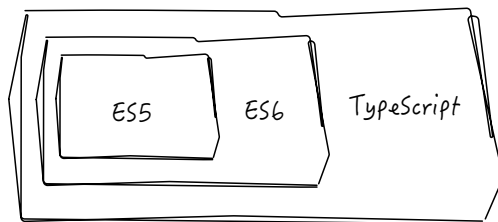


图 3-1 ES 5、ES 6 跟 TypeScript 的关系

在 JavaScript 这种弱类型语言中，简单自由（从另一种角度来说说是随意多变）的编写模式对开发者的技术水平要求较高，初学者跟资深开发者编写的代码质量可能差别很大，这不利于项目的维护。一方面，ES 6 引入了 `let` 变量声明和 `const` 常量声明、模板字符串、箭头函数、类、迭代器、生成器、模块和 `Promises` 等新特性，极大地增强了 JavaScript 语言的开发能力；另一方面，2009 年开始设计的 TypeScript 语言，经历了几年的发展后，最终向 ECMAScript 靠拢，实现了其标准，并在此基础上做了进一步增强，主要有类型校验、接口、装饰器等特性，这使得代码编写更规范化，也更有利于项目的维护。本章后续内容将会介绍这些增强特性。



本章并不会刻意去突出 ES 6 与 TypeScript 的异同，有 ES 6 基础的读者学习 TypeScript 成本会很低，甚至可以只关注接口、装饰器等部分内容即可。但 TypeScript 的核心是增强类型的处理，建议还是把所有知识点都学习一下，感受 TypeScript 带给我们的美妙编程体验。

3.1.2 安装

与 TypeScript 相关的工具一般是通过 `npm` 进行安装的。首先要查看 `npm` 是否已经安装，可以运行如下命令：

```
$ npm -v
```

接下来我们将使用 2.0 正式版本的 TypeScript，安装命令如下：

```
$ npm install -g typescript@2.0.0
```

安装完成后，编写第一个 TypeScript 程序，并保存到 `hello.ts` 文件中，文件的代码如下：

```
console.log('Hello TypeScript!');
```

在浏览器中运行 TypeScript 程序，必须先编译成浏览器能识别的 JavaScript 代码。可以通过 `tsc` 编译器来编译 TypeScript 文件，生成与之对应的 JavaScript 文件。编译过程如下：

```
$ tsc hello.ts
```

此时会在目录下面看到一个 `hello.js` 文件，该文件中的代码是基于 ES 3 / ES 5 标准的，能直接在浏览器中运行。

3.2 基本类型

在 TypeScript 中，提供了以下基本数据类型：

- 布尔类型（boolean）
- 数字类型（number）
- 字符串类型（string）
- 数组类型（array）
- 元组类型（tuple）
- 枚举类型（enum）
- 任意值类型（any）
- null 和 undefined
- void 类型
- never 类型

其中，元组、枚举、任意值、void 和 never 类型是 TypeScript 有别于 JavaScript 的特有类型。

在 TypeScript 中声明变量，需要加上类型声明，如 boolean 或 string 等。通过静态类型约束，在编译时执行类型检查，这样可以避免一些类型混用的低级错误。下面将具体介绍这些基本类型。

3.2.1 布尔类型

布尔类型是最简单的数据类型，只有 true 和 false 两种值。下面定义了一个布尔类型的变量 flag，并赋值为 true。由于 flag 被初始化为布尔类型，如果再赋值为非 boolean 的其他类型值，编译时会抛出错误。

```
let flag: boolean = true;
flag = 1; // 报错，不能把数字类型的值赋给布尔类型的变量。
```

3.2.2 数字类型

在 TypeScript 中，数字都是浮点型。TypeScript 同时支持二进制、八进制、十进制和十六进制字面量，示例代码如下：

```
let binaryLiteral: number = 0b1010; // 二进制
let octalLiteral: number = 0o744; // 八进制
```



```
let decliteral: number = 6; // 十进制
let hexliteral: number = 0xf00d; // 十六进制
```

3.2.3 字符串类型

TypeScript 支持使用单引号 (') 或双引号 (") 来表示字符串类型。除此之外, 还支持使用模板字符串反引号 (`) 来定义多行文本和内嵌表达式。使用 `\${ expr }` 的形式嵌入变量或表达式, 在处理拼接字符串的时候很有用。示例代码如下:

```
let name: string = "Angular";
let years: number = 5;
let words: string = `你好, 今年是 ${ name } 发布 ${ years + 1 } 周年`;
```

3.2.4 数组类型

TypeScript 数组的操作类似于 JavaScript 数组的操作, TypeScript 建议开发者最好只为数组元素赋一种类型的值。TypeScript 有两种数组定义方式, 示例代码如下:

```
// 在元素类型后面接上[]
let arr: number[] = [1, 2];
// 或者使用数组泛型
let arr: Array<number> = [1, 2];
```

3.2.5 元组类型

元组类型用来表示已知元素数量和类型的数组, 各元素的类型不必相同。下面定义了一个值分别为字符串和数字类型的元组。示例代码如下:

```
let x: [string, number];
x = ['Angular', 25]; // 运行正常
x = [10, 'Angular']; // 报错
console.log(x[0]); // 输出 Angular
```

3.2.6 枚举类型

枚举是一个可被命名的整型常数的集合, 枚举类型为集合成员赋予有意义的名称, 增强了可读性。示例代码如下:

```
enum Color {Red, Green, Blue};
let c: Color = Color.Blue;
console.log(c); // 输出: 2
```

枚举默认的下标是 0，可以手动修改默认的下标值。示例代码如下：

```
enum Color {Red = 2, Blue, Green = 6};  
let c: Color = Color.Blue;  
console.log(c); // 输出: 3
```

3.2.7 任意值类型

任意值是 TypeScript 针对编程时类型不明确的变量所使用的一种数据类型，它常用于以下三种情况。

- 当变量的值会动态变化时，比如来自用户的输入或第三方代码库，任意值类型可以让这些变量跳过编译阶段的类型检查。示例代码如下：

```
let x: any = 1; // 数字类型  
x = "I am a string"; // 字符串类型  
x = false; // 布尔类型
```

- 在改写现有代码时，任意值允许在编译时可选择地包含或移除类型检查。示例代码如下：

```
let x: any = 4;  
x.ifItExists(); // 正确，ifItExists方法在运行时可能存在，但是这里并不检查  
x.toFixed(); // 正确
```

- 在定义存储各种类型数据的数组时。示例代码如下：

```
let arrayList: any[] = [1, false, "fine"];  
arrayList[1] = 100;
```

3.2.8 null 和 undefined

在默认情况下，null 和 undefined 是其他类型的子类型，可以赋值给其他类型，如数字类型等，此时赋值后的类型会变成 null 或 undefined，致力于类型校验的 TypeScript 设计者们显然不希望这种类型变化给开发者带来额外的困扰。在 TypeScript 中启用 **严格的空校验 (--strictNullChecks)** 特性，就可以使得 null 和 undefined 只能被赋值给 void 或本身对应的类型。示例代码如下：

```
// 启用 --strictNullChecks  
let x: number;  
x = 1; // 运行正确  
x = undefined; // 运行错误  
x = null; // 运行错误
```

上面例子中变量 `x` 只能是数字类型。如果一个类型可能出现 `null` 或者 `undefined`，可以用 `|` 来支持多种类型。示例代码如下：

```
// 启用 --strictNullChecks
let x: number;
let y: number | undefined;
let z: number | null | undefined;
```

```
x = 1; // 运行正确
y = 1; // 运行正确
z = 1; // 运行正确
```

```
x = undefined; // 运行错误
y = undefined; // 运行正确
z = undefined; // 运行正确
```

```
x = null; // 运行错误
y = null; // 运行错误
z = null; // 运行正确
```

```
x = y; // 运行错误
x = z; // 运行错误
y = x; // 运行正确
y = z; // 运行错误
z = x; // 运行正确
z = y; // 运行正确
```

上面例子中变量 `y` 允许被赋予数字类型或 `undefined` 类型的数据值，而变量 `z` 还额外支持 `null`。TypeScript 官方建议在编码时，都启用 `--strictNullChecks` 特性，这样有利于编写更健壮的代码。

3.2.9 void 类型

在 TypeScript 中，使用 `void` 表示没有任何类型。例如，当一个函数没有返回值时，意味着返回值类型是 `void`。示例代码如下：

```
function hello(): void {
    alert("Hello Angular");
}
```

对于可忽略返回值的回调函数来说，使用 `void` 类型会比任意值类型更安全一些。示例代码如下：

```
function func(foo: () => void) {  
    let f = foo(); // 使用函数 foo 的返回值  
    f.doSth(); // 报错，void 类型不存在 doSth() 方法，此时换成任意值类型则不报错  
}
```

3.2.10 never 类型

`never` 是其他类型（包括 `null` 和 `undefined`）的子类型，代表从不会出现的值。这意味着声明为 `never` 类型的变量只能被 `never` 类型所赋值，在函数中它通常表现为抛出异常或无法执行到终止点（例如无限循环）。示例代码如下：

```
let x: never;  
let y: number;  
  
// 运行错误，数字类型不能转换为 never 类型  
x = 123;  
  
// 运行正确，never 类型可以赋值给 never 类型  
x = (() => { throw new Error('exception occur') })();  
  
// 运行正确，never 类型可以赋值给数字类型  
y = (() => { throw new Error('exception occur') })();  
  
// 返回值为 never 的函数可以是抛出异常的情况  
function error(message: string): never {  
    throw new Error(message);  
}  
  
// 返回值为 never 的函数可以是无法被执行到终止点的情况  
function loop(): never {  
    while (true) {  
    }  
}
```

3.3 声明和解构

在 TypeScript 中，支持 `var`、`let` 和 `const` 这样的声明方式。

3.3.1 let 声明

使用 let 与 var 声明变量的写法类似，示例代码如下：

```
let hello = "Hello Angular";
```

不同于 var，使用 let 声明的变量只在块级作用域内有效。示例代码如下：

```
function f(input: boolean) {  
  let a = 100;  
  if (input) {  
    let b = a + 1; // 运行正确  
    return b;  
  }  
  return b; // 错误，b 没有被定义  
}
```

这里定义了两个变量 a 和 b，a 的作用域是在 f() 函数体内，而 b 的作用域是在 if 语句块里。块级作用域还有一个问题，就是变量不能在它声明之前被读取或赋值。示例代码如下：

```
a++; // 错误，在声明之前使用是不合法的  
let a;
```

另外，要特别注意的是，在相同作用域中，let 不允许变量被重复声明。而在使用 var 声明时，不管声明几次，最后都只会得到最近一次声明的那个值。示例代码如下：

```
var x = 2;  
console.log( x + 3 ); // 输出：5  
var x = 3;  
console.log( x + 3 ); // 输出：6
```

```
let y = 2;  
let y = 3; // 报错，使用 let 声明的变量不能在一个作用域里多次声明
```

此外，还需要注意 let 声明在下面两种函数入参的对比：

```
function funA(x) {  
  let x = 100; // 报错，x 已经在函数入参声明  
}
```

```
// 增加了判断条件组成的新的块级作用域  
function funB(condition, x) {  
  if (condition) {
```

```
    let x = 100; // 运行正常
    return x;
  }
  return x;
}

funB(false, 0); // 返回 0
funB(true, 0); // 返回 100
```

3.3.2 const 声明

const 声明与 let 声明相似，它与 let 拥有相同的作用域规则，但 const 声明的是常量，常量不能被重新赋值，否则将编译错误。但是如果定义的常量是对象，对象里的属性值是可以被重新赋值的。示例代码如下：

```
const CAT_LIVES_NUM = 9;
const kitty = {
  name: "Aurora",
  numLives: CAT_LIVES_NUM
};

// 错误
kitty = {
  name: "Danielle",
  numLives: CAT_LIVES_NUM
};

kitty.name = "Kitty"; // 正确
kitty.numLives--; // 正确
```

3.3.3 解构

解构是 ES 6 的一个重要特性，TypeScript 在 1.5 版本中也开始增加了对解构的支持。所谓解构，就是将声明的一组变量与相同结构的数组或者对象的元素数值一一对应，并对变量相对应的元素进行赋值。解构可以帮助开发者非常容易地实现多返回值的场景，这样不仅写法简洁，也会增强代码的可读性。

在 TypeScript 中支持数组解构和对象解构，下面将对这两种不同的解构类型进行讲解。

数组解构

数组解构是最简单的解构类型，示例代码如下：

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // 相当于 input[0]: 1
console.log(second); // 相当于 input[1]: 2
```

也可作用于已声明的变量：

```
[first, second] = [second, first]; // 变量交换
```

或作用于函数参数：

```
function f([first, second]= [number, number]) {
  console.log(first + second);
}
```

```
f([1, 2]); // 输出：3
```

我们还可以在数组解构中使用 `rest` 参数语法（形式为“...变量名”）创建一个剩余变量列表，三个连续小数点“...”表示展开操作符，用于创建可变长的参数列表，使用起来非常方便。示例代码如下：

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // 输出：1
console.log(rest); // 输出：[ 2, 3, 4 ]
```

对象解构

对象解构有趣的地方是一些原本需要多行编写的代码，用对象解构的方式编写一行代码就能完成，代码很简洁、可读性强。示例代码如下：

```
let test = { x: 0, y: 10, width: 15, height: 20 };
let {x, y, width, height} = test;
console.log(x, y, width, height); // 输出：0,10,15,20
```

解构虽然很方便，但使用时还得多注意，特别是深层嵌套的场景，是比较容易出错的。

3.4 函数

不管什么编程语言，都少不了函数这个重要的概念，它用于定义特定的行为。TypeScript 在 JavaScript 函数的基础上添加了更多额外的功能，使函数变得更加易用。

3.4.1 函数定义

在 TypeScript 中支持函数声明和函数表达式的写法，示例代码如下：

// 函数声明写法

```
function maxA(x: number, y: number): number {  
    return x > y ? x : y;  
}
```

// 函数表达式写法

```
let maxB = function(x: number, y: number): number { return x > y ? x : y;};
```

在上例中，参数类型和返回值类型这两部分都是会被检查的。在调用时，只做参数类型和个数的匹配，不做参数名的校验。

3.4.2 可选参数

在 JavaScript 里，被调函数的每个参数都是可选的；而在 TypeScript 中，被调函数的每个参数都是必传的，在编译时，会检查函数的每个参数是否传值。简而言之，传递给一个函数的参数个数必须和函数定义时的参数个数一致。示例代码如下：

```
function max(x: number, y: number): number {  
    return x > y ? x : y;  
}
```

```
let result1 = max(2); // 报错  
let result2 = max(2, 4, 7); // 报错  
let result3 = max(2, 4); // 运行正常
```

但是经常会遇到根据实际需要来决定是否传入某个参数的情况，Typescript 提供了可选参数语法，即在参数名旁边加上“?”使其变成可选参数。示例代码如下：

```
function max(x: number, y?: number): number {  
    if (y)  
        return x > y ? x : y;  
    } else {
```



```
    return x;
  }
}

let result1 = max(2); // 运行正常
let result2 = max(2, 4, 7); // 报错
let result3 = max(2, 4); // 运行正常
```



需要注意的是，可选参数必须位于必选参数的后面。

3.4.3 默认参数

TypeScript 还支持初始化默认参数。如果函数的某个参数设置了默认值，当该函数被调用时，如果没有给这个参数传值或者传的值为 `undefined`，这个参数的值就是设置的默认值。示例代码如下：

```
function max(x: number, y = 4): number {
  return x > y ? x : y;
}

let result1 = max(2); // 运行正常
let result2 = max(2, undefined); // 运行正常
let result3 = max(2, 4, 7); // 报错
let result4 = max(2, 4); // 运行正常
```

带默认值的参数不必放在必选参数的后面，但如果带默认值的参数放到了必选参数的前面，用户必须显式地传入 `undefined`。示例代码如下：

```
function max(x=2, y: number): number {
  return x > y ? x : y;
}

let result1 = max(2); // 报错
let result2 = max(undefined, 4); // 运行正常
let result3 = max(2, 4, 7); // 报错
let result4 = max(2, 4); // 运行正常
```

3.4.4 剩余参数

上面介绍了函数中的必选参数、默认参数及可选参数，它们的共同点是只能表示某一个参数。当需要同时操作多个参数，或者并不知道会有多少个参数传递进来时，就需要用到 TypeScript 里的剩余参数了。在 TypeScript 里，所有的可选参数都可以放到一个变量里。示例代码如下：

```
function sum(x:number, ...restOfNumber:number[]): number {  
    let result = x;  
    for(let i = 0; i < restOfNumber.length; i++){  
        result += restOfNumber[i];  
    }  
    return result;  
}  
let result = sum(1, 2, 3, 4, 5);  
console.log(result); // 输出: 15
```



需要注意的是，剩余参数可以理解为个数不限的可选参数，即剩余参数包含的参数个数可以为零到多个。

3.4.5 函数重载

函数重载通过为同一个函数提供多个函数类型定义来达到实现多种功能的目的。TypeScript 支持函数重载，示例代码如下：

```
function css(config: {});  
function css(config: string, value: string);  
function css(config: any, value?: any) {  
    if (typeof config === 'string') {  
        // ...  
    } else if (typeof config === 'object') {  
        // ...  
    }  
}
```

在上面的例子中，css 函数有三个重载函数，编译器会根据参数类型来判断该调用哪个函数。TypeScript 的函数重载通过查找重载列表来实现匹配，根据定义的优先顺序来依次匹配，所以在实现重载函数时，建议把最精确的定义放在最前面。

3.4.6 箭头函数

JavaScript 的 `this` 是一个重要的概念，学习 TypeScript 有必要弄清楚 `this` 的工作机制，这能帮助我们避免一些隐蔽的 bug。例如下面的这段“问题”代码：

```
let gift = {
  gifts: ["teddy bear", "spiderman", "dinosaur", "Thomas loco", "toy bricks", "
    Transformers"],
  giftPicker: function() {
    return function() {
      let pickedNumber = Math.floor(Math.random() *6);
      return this.gifts[pickedNumber];
    }
  }
}

let pickGift = gift.giftPicker();
console.log("you get a : " + pickGift());
```

运行程序，发现并不能输出预期的结果，而是抛出以下错误：

```
Uncaught TypeError: Cannot read property '5' of undefined(...)
```

这是因为 `giftPicker()` 函数里的 `this` 被设置成了 `window` 而不是 `gift` 对象。因为这里没有对 `this` 进行动态绑定，因此 `this` 就指向了 `window` 对象。

TypeScript 提供的箭头函数 (`=>`) 很好地解决了这个问题，它在函数创建时就绑定了 `this`，而不是在函数调用时。示例代码如下：

```
let gift = {
  gifts: ["teddy bear", "spiderman", "dinosaur", "Thomas loco", "toy bricks", "
    Transformers"],
  giftPicker: function() {
    return () => {
      let pickedNumber = Math.floor(Math.random() *6);
      return this.gifts[pickedNumber];
    }
  }
}

let pickGift = gift.giftPicker();
console.log("you get a : " + pickGift());
```

3.5 类

传统的 JavaScript 程序使用函数和基于原型（Prototype）的继承来创建可重用的“类”，这对于习惯了面向对象编程的开发者来说不是很友好。好在 TypeScript 支持使用基于类的面向对象编程。

3.5.1 类的例子

下面看一个定义类的例子。示例代码如下：

```
class Car {  
  engine: string;  
  constructor(engine: string) {  
    this.engine = engine;  
  }  
  drive(distanceInMeters: number = 0) {  
    console.log( `A car runs ${distanceInMeters}m powered by ` + this.engine);  
  }  
}
```

上面声明了一个汽车类 Car，这个类有三个类成员：类属性 engine、构造函数及 drive() 方法，其中类属性 engine 可通过 this.engine 访问。下面实例化一个 Car 的新对象，并执行构造函数初始化。

```
let car = new Car("petrol");
```

调用成员方法并输出结果：

```
car.drive(100); // 输出：A car runs 100m powered by petrol
```

3.5.2 继承与多态

封装、继承、多态是面向对象的三大特性。在上面的例子中把汽车的行为写到一个类中，即所谓的封装。在 TypeScript 中，使用 extends 关键字即可方便地实现继承。示例代码如下：

```
// 继承前文的 Car 类  
class MotoCar extends Car {  
  constructor(engine: string) { super(engine); }  
}
```

```
class Jeep extends Car {
  constructor(engine: string) { super(engine); }
  drive(distanceInMeters: number = 100) {
    console.log("Jeep...");
    return super.drive(distanceInMeters);
  }
}

let tesla = new MotoCar("electricity");
let landRover: Car = new Jeep("petrol"); // 实现多态

tesla.drive(); // 调用父类的 drive() 方法
landRover.drive(200); // 调用子类的 drive() 方法
```

从上面的例子可以看到，MotoCar 和 Jeep 是基类 Car 的子类，通过 extends 来继承父类，子类可以访问父类的属性和方法，也可以重写父类的方法。Jeep 的 drive() 方法重写了 Car 的 drive() 方法，这样 drive() 方法在不同的类中就具有不同的功能，这就是多态。



即使 landRover 被声明为 Car 类型，它也依然是子类 Jeep，landRover.drive(200) 调用的是 Jeep 里的重写方法。派生类构造函数必须调用 super()，它会执行基类的构造方法。

3.5.3 修饰符

在类中的修饰符可以分为公共（public）、私有（private）和受保护（protected）三种类型。

public 修饰符

在 TypeScript 里，每个成员默认为 public，可以被自由地访问。我们可以显式地给 Car 类里定义的成员加上 public 修饰符，示例代码如下：

```
class Car {
  public engine: string;
  public constructor(engine: string) {
    this.engine = engine;
  }
}
```

```
public drive(distanceInMeters: number = 0) {  
    console.log( `A car runs ${distanceInMeters}m powered by ` + this.engine);  
}  
}
```

private 修饰符

当类成员被标记成 `private` 时，就不能在类的外部访问它了。示例代码如下：

```
class Car {  
    private engine: string;  
    constructor(engine: string) {  
        this.engine = engine;  
    }  
}
```

`new Car("petrol").engine;` // 报错: `engine` 属性是私有的，只能在类内部访问



ES 6 并没有提供对私有属性的语法支持，但是可以通过闭包来实现私有属性。

protected 修饰符

`protected` 修饰符与 `private` 修饰符的行为很相似，但有一点不同，`protected` 成员在派生类中仍然可以访问。示例代码如下：

```
class Car {  
    protected engine: string;  
    constructor(engine: string) {  
        this.engine = engine;  
    }  
    drive(distanceInMeters: number = 0) {  
        console.log( `A car runs ${distanceInMeters}m powered by ` + this.engine);  
    }  
}  
  
class MotoCar extends Car {  
    constructor(engine: string) { super(engine); }  
    drive(distanceInMeters: number = 50) {
```

```

        super.drive(distanceInMeters);
    }
}

let tesla = new MotoCar("electricity");
// 运行正常，输出：A car runs 50m powered by electricity
console.log(tesla.drive());
// 报错
console.log(tesla.engine);

```



由于 `engine` 被声明为 `protected`，所以不能在外部访问它，但是仍然可以通过它的继承类 `MotoCar` 来访问。

3.5.4 参数属性

参数属性是通过给构造函数参数添加一个访问限定符（`public`、`protected` 及 `private`）来声明的。参数属性可以方便地让我们在一个地方定义并初始化类成员。使用参数属性对上述 `Car` 类进行改造，示例代码如下：

```

class Car {
  constructor(protected engine: string) {}
  drive(distanceInMeters: number = 0) {
    console.log(`A car runs ${distanceInMeters}m powered by ` + this.engine);
  }
}

```

在构造函数里通过 `protected engine: string` 来创建和初始化 `engine` 成员属性，从而把声明和赋值合并到一处。

3.5.5 静态属性

类的静态成员存在于类本身而不是类的实例上，类似于在实例属性上使用“`this.`”来访问属性，我们使用“`类名.`”来访问静态属性。可以使用 `static` 关键字来定义类的静态属性，示例代码如下：

```

class Grid {
  static origin = {x: 0, y: 0};
  constructor (public scale: number) { }
  calculateDistanceFromOrigin(point: {x: number; y: number;}) {

```

```
    let xDist = (point.x - Grid.origin.x);
    let yDist = (point.y - Grid.origin.y);
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }
}

let grid1 = new Grid(1.0);
let grid2 = new Grid(5.0);

// 输出: 14.142135623730951
console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
// 输出: 2.8284271247461903
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

3.5.6 抽象类

TypeScript 有抽象类的概念，它是供其他类继承的基类，不能被实例化。不同于接口，抽象类必须包含一些抽象方法，同时也可以包含非抽象的成员。`abstract` 关键字用于定义抽象类和抽象方法。抽象类中的抽象方法不包含具体实现，并且必须在派生类中实现。示例代码如下：

```
abstract class Person {
  abstract speak(): void; // 必须在派生类中实现
  walking(): void {
    console.log('Walking on the road');
  }
}

class Male extends Person {
  speak(): void {
    console.log('How are you?');
  }
}

let person: Person; // 创建一个抽象类引用
person = new Person(); // 报错: 不能创建抽象类实例
person = new Male(); // 创建一个 Male 实例
person.speak();
person.walking();
```




在面向对象设计中，抽象类和接口是经常讨论的话题，在 TypeScript 中也一样。简单来说，接口更注重功能的设计，抽象类更注重结构内容的体现。

3.6 模块

ES 6 引入了模块的概念，在 TypeScript 中也支持模块的使用。

3.6.1 概述

模块是自声明的，两个模块之间的关系是通过在文件级别上使用 `import` 和 `export` 来建立的。TypeScript 与 ES 6 一样，任何包含顶级 `import` 或者 `export` 的文件都会被当成一个模块。

模块在其自身的作用域里执行，而不是在全局作用域里，这意味着定义在一个模块里的变量、函数和类等，在模块外部是不可见的，除非明确地使用 `export` 导出它们。类似的，如果想使用其他模块导出变量、函数、类和接口，则必须先通过 `import` 导入它们。

模块使用模块加载器来导入它的依赖，模块加载器在代码运行时 would 查找并加载模块间的所有依赖。在 Angular 中，常用的模块加载器有 SystemJS 和 Webpack。

3.6.2 模块导出方式

模块可以通过导出的方式来提供变量、函数、类、类型别名和接口给外部模块调用，导出的方式分为以下三种。

- 导出声明

任何模块都能够通过 `export` 关键字导出，示例代码如下：

```
export const COMPANY = "GF"; // 导出变量

export interface IdentityValidate { // 导出接口
  isGfStaff(s: string): boolean;
}

export class ErpIdentityValidate implements IdentityValidate { // 导出类
  isGfStaff(erp: string) {
    return erpService.contains(erp); // 判断是否为内部员工
  }
}
```

- 导出语句

当需要对导出的模块进行重命名时，就用到了导出语句。示例代码如下：

```
class ErpIdentityValidate implements IdentityValidate { // 导出类
  isGfStaff(erp: string) {
    return erpService.contains(erp);
  }
}
```

```
export { ErpIdentityValidate };
export { ErpIdentityValidate as GfIdentityValidate };
```

- 模块包装

有时候我们需要修改和扩展已有的模块，并导出供其他模块调用，这时可以使用模块包装来再次导出。示例代码如下：

```
// 导出原先的验证器，但做了重命名
export { ErpIdentityValidate as RegExpBasedZipCodeValidator } from
  "./ErpIdentityValidate";
```

一个模块可以包裹多个模块，并把新的内容以一个新的模块导出。示例代码如下：

```
export * from "./IdentityValidate";
export * from "./ErpIdentityValidate";
```

3.6.3 模块导入方式

模块导入与模块导出相对应，可以使用 `import` 关键字来导入当前模块依赖的外部模块。导入有如下两种方式。

- 导入一个模块

```
import { ErpIdentityValide } from "./ErpIdentityValidate";
let erpValide = new ErpIdentityValidate();
```

- 别名导入

```
import { ErpIdentityValidate as ERP } from "./ErpIdentityValidate";
let erpValidator = new ERP();
```

另外，我们也可以对整个模块进行别名导入，将整个模块导入到一个变量中，并通过它来访问模块的导出部分。示例代码如下：

```
import * as validator from "./ErpIdentityValidate";  
let myValidate = new validator.ErpIdentityValidate();
```

3.6.4 模块的默认导出

模块可以用 `default` 关键字实现默认导出的功能，每个模块都可以有一个默认导出。类和函数声明可以直接省略导出名来实现默认导出。默认导出有利于减少调用方调用模块的层数，省去一些冗余的模块前缀书写。接下来看看几类默认导出的例子。

- 默认导出类

```
// ErpIdentityValidate.ts  
export default class ErpIdentityValidate implements IdentityValidate {  
  isGfStaff(erp: string) {  
    return erpService.contains(erp);  
  }  
}
```

```
// test.ts  
import validator from "./ErpIdentityValidate";  
let erp = new validator();
```

- 默认导出函数

```
// nameServiceValidate.ts  
export default function (s: string) {  
  return nameService.contains(s);  
}
```

```
// test.ts  
import validate from "./nameServiceValidate";  
let name = "Angular";
```

```
// 使用导出函数  
console.log(`"${name}" ${validate(name) ? " matches" : " does not match"}`);
```

- 默认导出值

```
// constantService.ts  
export default "Angular";
```

```
// test.ts  
import name from "./constantService";  
console.log(name); // "Angular"
```

3.6.5 模块设计原则

在模块设计中，共同遵循一些原则有利于更好地编写和维护项目代码。下面列出几点模块设计的原则。

尽可能在顶层导出

顶层导出可以降低调用方使用的难度，过多的“.”操作使得开发者要记住过多的细节，所以尽量使用默认导出或者在顶层导出。单个对象（类或函数等）可以采用默认导出的方式：

```
// ClassTest.ts
export default class ClassTest {
  // ...
}

// FuncTest.ts
export default function FuncTest() {
  // ...
}

// Test.ts
import ClassTest from "./ClassTest";
import FuncTest from "./FuncTest";

let C = new ClassTest();
FuncTest();
```

但是，如果要返回多个对象，则可以采用在顶层导出的方式，调用时再明确地列出所导入的对象名称即可。示例代码如下：

```
// ModuleTest.ts
export class ClassTest {
  // ...
}
export function FuncTest() {
  // ...
}

// Test.ts
import { ClassTest, FuncTest } from "./ModuleTest";
```

```
let C = new ClassTest();
FuncTest();
```

明确列出导入对象的名称

在导入时尽可能明确地指定导入对象的名称，这样只要接口不变，调用方式就可以不变，从而降低了导入跟导出模块的耦合度，做到面向接口编程。

```
import { ClassTest, FuncTest } from "./ModuleTest";
```

使用命名空间模式导出

```
// MyLargeModule.ts
export class Dog { /* ... */ }
export class Cat { /* ... */ }
export class Tree { /* ... */ }
export class Flower { /* ... */ }

// Consumer.ts
import * as myLargeModule from "./MyLargeModule";
let x = new myLargeModule.Dog();
```

使用模块包装进行扩展

我们可能经常需要扩展一个模块的功能，推荐的方案是不要去改变原来的对象，而是导出一个新的对象来提供新的功能。示例代码如下：

```
// ModuleA.ts
export class ModuleA {
  constructor() { /* ... */ }
  sayHello() {
    // ...
  }
}

// ModuleB.ts
import { ModuleA } from "./ModuleA";
class ModuleB extends ModuleA {
  constructor() { super(); /* ... */ }
```

```
// 添加新的方法
sayHi() {
  // ...
}
}
export { ModuleB as ModuleA };

// Test.ts
import { ModuleA } from "./ModuleB";
let C = new ModuleA();
```

3.7 接口

3.7.1 概述

接口在面向对象设计中具有极其重要的作用，在 GoF 的 23 种设计模式中，基本上都可见到接口的身影。长期以来，接口模式一直是 JavaScript 这类弱类型语言的软肋，虽然有类似于“鸭式辨型”等的各种伪实现，并有诸如《JavaScript 设计模式》等书籍的问世，但使用起来还是略为烦琐。TypeScript 接口的使用方式类似于 Java，同时还增加了更灵活的接口类型，包括属性、函数、可索引（Indexable Type）和类等类型。

3.7.2 属性类型接口

在 TypeScript 中，使用 interface 关键字来定义接口。下面通过一个简单示例来了解属性接口。示例代码如下：

```
interface FullName {
  firstName: string;
  secondName: string;
}

function printLabel(name: FullName) {
  console.log(name.firstName + " " + name.secondName);
}

let myObj = {age: 10, firstName: "Jim", secondName: "Raynor"};
printLabel(myObj);
```

上例中接口 FullName 包含两个属性：firstName 和 secondName，且都为字符串类型。这里有两点需要注意：

- 传给 `printLabel()` 方法的对象只要“形式上”满足接口的要求即可。例如，上例中对象 `myObj` 必须包含 `firstName` 和 `secondName` 属性，且类型都为 `string`。
- 接口类型检查器不会去检查属性的顺序，但要确保相应的属性存在且类型匹配。

TypeScript 还提供了可选属性，可选属性对可能存在的属性进行预定义，并兼容不传值的情况。带有可选属性的接口与普通接口的定义方式差不多，区别是在定义的可选属性变量名后面加一个“?”符号。示例代码如下：

```
interface FullName {  
  firstName: string;  
  secondName?: string;  
}  
  
function printLabel(name:FullName) {  
  console.log(name.firstName + " " + name.secondName);  
}
```

```
let myObj = {firstName: "Jim"}; // secondName 是可选属性，可以不传  
printLabel(myObj); // 输出: Jim undefined
```

3.7.3 函数类型接口

接口除可以描述带有属性的普通对象外，也能描述函数类型。在定义函数类型接口时，需要明确定义函数的参数列表和返回值类型，且参数列表中的每个参数都要有参数名和类型。示例代码如下：

```
interface encrypt {  
  (val:string, salt:string):string  
}
```

定义好函数类型接口 `encrypt` 之后，接下来将通过一个例子来展示如何使用函数类型接口。示例代码如下：

```
let md5: encrypt;  
md5 = function(val:string, salt:string){  
  console.log("origin value:" + val);  
  let encryptValue = doMd5(val,salt); // doMd5只是一个 mock 方法  
  console.log("encrypt value:" + encryptValue);  
  return encryptValue;  
}  
let pwd = md5("password","Angular");
```

对于函数类型接口要注意下面两点。

- 函数的参数名：使用时参数个数需要与接口定义的参数个数相同，对应位置变量的数据类型需要保持一致，参数名可以不一样。
- 函数返回值：函数的返回值类型与接口定义的返回值类型要一致。

3.7.4 可索引类型接口

可索引类型接口用来描述那些可以通过索引得到的类型，比如 `userArray[1]`、`userObject["name"]` 等。它包含一个索引签名，表示用来索引的类型与返回值类型，即通过特定的索引来得到指定类型的返回值。示例代码如下：

```
interface UserArray {  
  [index: number]: string;  
}  
interface UserObject {  
  [index: string]: string;  
}  
  
let userArray: UserArray;  
let userObject: UserObject;  
  
userArray = ["张三", "李四"];  
userObject = {"name": "张三"};  
  
console.log(userArray[0]); // 输出：张三  
console.log(userObject["name"]); // 输出：张三
```

索引签名支持字符串和数字两种数据类型。使用这两种类型的最终返回值可以是一样的，即当使用数字类型来索引时，JavaScript 最终也会将它转换成字符串类型后再去索引对象。比如在上例中，以下写法最终输出的结果是相同的。

```
// ...  
console.log(userArray[0]); // 输出：张三  
console.log(userArray["0"]); // 输出：张三
```

3.7.5 类类型接口

类类型接口用来规范一个类的内容。示例代码如下：


```
interface Animal {  
    name: string;  
}  
  
class Dog implements Animal {  
    name: string;  
    constructor(n: string) { }  
}
```

我们可以在接口中描述一个方法,并在类里具体实现它的功能,如同下面的 setName 方法一样:

```
interface Animal {  
    name: string;  
    setName(n: string): void;  
}  
  
class Dog implements Animal {  
    name: string;  
    setName(n: string) {  
        this.name = n;  
    }  
    constructor(n: string) { }  
}
```

3.7.6 接口扩展

和类一样,接口也可以实现相互扩展,即将成员从一个接口复制到另一个接口里面,这样可以更灵活地将接口拆分到可复用的模块里。示例代码如下:

```
interface Animal {  
    eat(): void;  
}  
  
interface Person extends Animal {  
    talk(): void;  
}  
  
class Programmer {  
    coding():void {
```

```
    console.log('wow, TypeScript is the best language');  
  }  
}
```

```
class ITGirl extends Programmer implements Person{  
  eat(){  
    console.log('animal eat');  
  }  
  
  talk(){  
    console.log('person talk');  
  }  
  
  coding():void {  
    console.log('I am a girl, but i like coding.');  }  
}
```

```
let itGirl = new ITGirl(); // 通过组合集成类实现接口扩展，可以更灵活地复用模块  
itGirl.coding();
```

3.8 装饰器

3.8.1 概述

装饰器（Decorator）是一种特殊类型的声明，它可以被附加到类声明、方法、属性或参数上。装饰器由 `@` 符号紧接一个函数名称表示，形如 `@expression`，`expression` 求值后必须是一个函数，在函数执行时装饰器的声明方法会被执行。正如名字所示，装饰器是用来给附着的主体进行装饰，添加额外的行为的。

在 TypeScript 的源码中，可以看到官方提供了如下几种类型的装饰器。

// 方法装饰器

```
declare type MethodDecorator = <T>(target: Object, propertyKey: string | symbol,  
  descriptor: TypedPropertyDescriptor<T>) => TypedPropertyDescriptor<T> | void;
```

// 类装饰器

```
declare type ClassDecorator = <TFunction extends Function>(target: TFunction) =>  
  TFunction | void;
```

```
// 参数装饰器
declare type ParameterDecorator = (target: Object, propertyKey: string | symbol,
    parameterIndex: number) => void;

// 属性装饰器
declare type PropertyDecorator = (target: Object, propertyKey: string | symbol) =>
    void;
```

如上所示，每种装饰器类型传入的参数不大相同，下面将分别进行介绍。

3.8.2 方法装饰器

方法装饰器是在声明一个方法之前被声明的（紧贴着方法声明），它会被应用到方法的属性描述符上，可以用来监视、修改或替换方法定义。方法装饰器的声明如下：

```
declare type MethodDecorator = <T>(target: Object, propertyKey: string | symbol,
    descriptor: TypedPropertyDescriptor<T>) => TypedPropertyDescriptor<T> | void;
```

方法装饰器表达式在运行时会被当作函数来调用，传入下列三个参数。

- target: 类的原型对象。
- propertyKey: 方法的名字。
- descriptor: 成员属性描述。

其中，descriptor 的类型为 TypedPropertyDescriptor，在 TypeScript 中定义如下：

```
interface TypedPropertyDescriptor<T> {
    enumerable?: boolean; // 是否可遍历
    configurable?: boolean; // 属性描述是否可改变或属性是否可删除
    writable?: boolean; // 是否可修改
    value?: T; // 属性的值
    get?: () => T; // 属性的访问器函数（getter）
    set?: (value: T) => void; // 属性的设置器函数（setter）
}
```



想了解更多关于 descriptor 的内容，可以到 MDN 查看更多关于 Object.defineProperty() 的介绍。

接下来看一个方法装饰器的例子，示例代码如下：

```
class TestClass {  
  @log  
  testMethod(arg: string) {  
    return "logMsg: " + arg;  
  }  
}
```

下面是方法装饰器 @log 的实现。

```
function log(target: Object, propertyKey: string, descriptor:  
  TypedPropertyDescriptor<any>) {  
  let origin = descriptor.value;  
  descriptor.value = function(...args: any[]) {  
    console.log("args: " + JSON.stringify(args)); // 调用前  
    let result = origin.apply(this, args); // 调用方法  
    console.log("The result-" + result); // 调用后  
    return result; // 返回结果  
  };  
  
  return descriptor;  
}
```

然后可以使用以下代码进行测试。

```
new TestClass().testMethod("test method decorator");
```

结果输出如下：

```
args: ["test method decorator"]  
The result-logMsg: test method decorator
```

当方法装饰器 @log 被调用时，它会打印日志信息。

3.8.3 类装饰器

类装饰器是在声明一个类之前被声明的，它应用于类构造函数，可以用来监视、修改或替换类定义。在 TypeScript 中定义如下：

```
declare type ClassDecorator = <TFunction extends Function>(target: TFunction) =>  
  TFunction | void;
```

如上所示，类的构造函数作为其唯一的参数。类装饰器在运行时会被当作函数的形式来调用。

假如类装饰器返回了一个值，那么它会在构造函数中替换类的声明。下面是使用类装饰器 (@Component) 的例子，应用到 Person 类。示例代码如下：

```
@Component({
  selector: 'person',
  template: 'person.html'
})
class Person {
  constructor(
    public firstName:string,
    public secondName:string
  ){}
}
```

关于类装饰器 @Component 的定义如下：

```
function Component(component) {
  return (target:any) => {
    return componentClass(target, component);
  }
}

// componentClass 的实现
function componentClass(target:any, component?:any):any {
  var original = target;
  function construct(constructor, args) { // 处理原型链
    let c:any = function () {
      return constructor.apply(this, args);
    };
    c.prototype = constructor.prototype;
    return new c;
  }

  let f:any = (...args) => { // 打印参数
    console.log("selector:" + component.selector);
    console.log("template:" + component.template);
    console.log(`Person: ${original.name}(${JSON.stringify(args)})`);
    return construct(original, args);
  };
};
```

```
f.prototype = original.prototype;
return f; // 返回构造函数
}
```

然后可以使用以下代码进行测试。

```
let p = new Person("Angular", "JS");
```

结果输出如下：

```
selector:person
template:person.html
Person: Person(["Angular","JS"])
```

如上所示，代码看起来有点烦琐，因为返回了一个新的构造函数，必须自己处理好原来的原型链。

3.8.4 参数装饰器

参数装饰器是在声明一个参数之前被声明的，它应用于类的构造函数或方法声明。参数装饰器在运行时会被当作函数的形式来调用，定义如下：

```
declare type ParameterDecorator = (target: Object, propertyKey: string | symbol,
parameterIndex: number) => void;
```

如上所述，包含三个参数。

- **target**：对于静态成员来说是类的构造函数，对于实例成员来说是类的原型对象。
- **propertyKey**：参数名称。
- **parameterIndex**：参数在函数参数列表中的索引。

下面是参数装饰器的一个简单例子。

```
class userService {
  login(@inject name: string) {}
}

// @inject 装饰器的实现
function inject(target: any, propertyKey: string | symbol, parameterIndex: number)
{
  console.log(target); // userService prototype
  console.log(propertyKey);
  console.log(parameterIndex);
}
```

结果输出如下：

```
Object  
login  
0
```



参数装饰器在 Angular 中被广泛使用，特别是结合 `reflect-metadata` 库来支持实验性的 Metadata API，读者可到官网了解更多相关知识。

3.8.5 属性装饰器

属性装饰器的定义如下：

```
declare type PropertyDecorator = (target: Object, propertyKey: string | symbol) =>  
    void;
```

如上所述，包含两个参数。

- `target`：对于静态成员来说是类的构造函数，对于实例成员来说是类的原型对象。
- `propertyKey`：属性名称。

属性装饰器是用来修饰类的属性的，其声明和被调用方式跟其他装饰器类似，具体内容不展开细讲了。

3.8.6 装饰器组合

TypeScript 支持多个装饰器同时应用到一个声明上，实现多个装饰器的复合使用，语法可以从左到右书写，示例如下：

```
@decoratorA @decoratorB param
```

或从上到下书写：

```
@decoratorA  
@decoratorB  
functionA
```

在 TypeScript 中，当多个装饰器应用在同一个声明上时，会进行如下步骤的操作：

- 从左到右（从上到下）依次执行装饰器函数，得到返回结果。
- 返回结果会被当作函数，从左到右（从上到下）依次调用。

下面是两个类装饰器复合应用的例子，注意看输出结果所显示的执行顺序。示例代码如下：

```
function Component(component) {  
  console.log('selector: ' + component.selector);  
  console.log('template: ' + component.template);  
  console.log('component init');  
  return (target: any) => {  
    console.log('component call');  
    return target;  
  }  
}
```

```
function Directive(directive) {  
  console.log('directive init');  
  return (target: any) => {  
    console.log('directive call');  
    return target;  
  }  
}
```

```
@Component({  
  selector: 'person',  
  template: 'person.html'  
})  
@Directive()  
class Person {  
}
```

```
let p = new Person();
```

结果输出如下：

```
selector: person  
template: person.html  
component init  
directive init  
directive call  
component call
```




Angular 框架在依赖注入、组件等部分有多个装饰器复合应用的场景，读者在本书后续的学习中可以进一步掌握这部分知识。

装饰器是 ES 7 的草案标准，在 Angular 中，装饰器一般会借助于元数据（Metadata）来定义组件。因此，读者掌握了装饰器的原理后，可以实现类似于 Angular 装饰器的语法糖。

3.9 泛型

在实际开发中，对于我们定义的 API，不仅仅需要考虑功能是否健全，还需要考虑它的复用性，更多的时候需要支持不特定的数据类型，而泛型（Generic）就是用来实现这样的效果的。

比如有一个最小堆算法，需要同时支持数字和字符串两种类型，可以通过把集合类型改为任意值类型（any）来实现，但是这样就等于放弃了类型检查，而我们希望的是返回的类型和参数类型一致。示例代码如下：

```
class MinHeap<T> {  
  list: T[] = [];  
  
  add(element: T): void {  
    // 这里进行大小比较，并将最小值放在数组头部  
  }  
  
  min(): T {  
    return this.list.length ? this.list[0] : null;  
  }  
}  
  
let heap1 = new MinHeap<number>();  
heap1.add(3);  
heap1.add(5);  
console.log(heap1.min());  
  
let heap2 = new MinHeap<string>();  
heap2.add('a');  
heap2.add('c');
```

```
console.log(heap2.min());
```

在上面的例子中分别声明了一个适用于数字类型和字符串类型的最小堆实例，给 `MinHeap` 类增加了类型变量 `T`，用于帮助捕获用户输入的数据类型，以便于后边的跟踪和使用。

泛型也支持函数，下面实现的 `zip` 函数用于把两个数组压缩到一起，其中声明了两个泛型类型 `T1` 和 `T2`。示例代码如下：

```
function zip<T1, T2>(l1: T1[], l2: T2[]): [T1, T2][] {
    let len = Math.min(l1.length, l2.length);
    let ret = [];
    for (let i = 0; i < len; i++) {
        ret.push([l1[i], l2[i]]);
    }
    return ret;
}

console.log(zip<number, string>([1,2,3], ['Jim', 'Sam', 'Tom']));
```



注意不要定义一个从未使用过的其他类型参数的泛型类型。

3.10 TypeScript 周边

3.10.1 编译配置文件

`tsc` 编译器有很多命令行参数，都写在命令行上会十分烦琐。`tsconfig.json` 文件正是用来解决这个问题的，它使得编译参数能在文件中维护。当运行 `tsc` 时，编译器从当前目录向上搜索 `tsconfig.json` 文件来加载配置，类似于 `package.json` 文件的搜索方式。

我们可以从一个空的 `tsconfig.json` 文件开始配置，如下所示：

```
{}
```

接着，可以往里面添加更复杂的配置参数，配置文件如下：

```
{
  "compilerOptions": {
    "target": "es5",
```

```
"module": "commonjs",
"declaration": false,
"noImplicitAny": false,
"removeComments": true,
"noLib": false,
"emitDecoratorMetadata": true,
"experimentalDecorators": true,
"sourceMap": true
},
"exclude": [
  "node_modules",
  "typings/browser.d.ts",
  "typings/browser/**"
],
"compileOnSave": false
}
```



在上述配置文件中，各项配置的详细说明可到官网查看，这里不再赘述，只是举例说明通过配置文件能减少烦琐的命令行参数，提升开发效率，方便开发维护。

3.10.2 声明文件

JavaScript 语言本身并没有静态类型检查的功能，而 TypeScript 编译器也只提供了 ECMAScript 标准里的标准库类型声明，只能识别 TypeScript 代码中的类型。如果想引用第三方的 JavaScript 库如 jQuery、lodash 等，就需要使用声明文件（Declaration File）来辅助开发。在 TypeScript 中，声明文件是以 `.d.ts` 为后缀的形式存在的，其作用是描述一个 JavaScript 模块文件所有导出的接口类型信息。

从 TypeScript 2.0 开始，可以不使用 typings，而直接使用 npm 来获取声明文件，命令行代码如下：

```
npm install --save @types/lodash
```

在上面例子中，`@types/lodash` 这个包实际上是来源于 DefinitelyTyped (<https://github.com/DefinitelyTyped/DefinitelyTyped>) 这个声明文件集合的项目。下载成功后就可以在 TypeScript 代码中使用该模块了。我们可以选择使用模块的形式导入，示例代码如下：

```
import * as _ from "lodash";
```

```
_.padStart("Hello Angular!", 2, " ");
```

如果配置了 `tsconfig.json` 文件，也可以直接使用全局变量 “_”，示例代码如下：

```
_.padStart("Hello Angular!", 2, " ");
```

在 `tsconfig.json` 文件中可以使用 `typeRoots` 来指定允许查找的包的文件夹路径，配置示例如下：

```
{
  "compilerOptions": {
    "typeRoots" : ["/typings"]
  }
}
```

上面的配置表示编译器会去查找所有 “./typings” 目录下的包，但不包含 `./node_modules/@types` 里面的包。

如果在 `tsconfig.json` 中配置了 `types`，那么只有配置的包才会被包含进来。配置示例如下：

```
{
  "compilerOptions": {
    "types" : [ "lodash", "koa" ]
  }
}
```



另外，可以通过设置 `"types": []`（即设置 `types` 为空数组）来禁止自动引入 `@type` 的包（在全局变量的模式下）。

3.10.3 编码工具

优秀的编码工具提供了类似于智能提示、查找引用及查找定义等功能，这样就可以大大地提高开发者的编程效率。下面介绍几款适合 TypeScript 的优秀编码工具，以飨读者。

- VS Code：微软官方推荐，TypeScript 集成度最好，免费。
- Atom：Github 推出的 IDE，可安装 `atom-typescript`（<https://atom.io/packages/atom-typescript>）插件支持 TypeScript 开发，免费。

- Sublime：轻量级，结合插件可以打造强大舒适的开发工具，可以下载 TypeScript-Sublime-Plugin (<https://github.com/Microsoft/TypeScript-Sublime-Plugin>) 插件开发 TypeScript，免费。
- WebStorm：前端开发功能最齐全、最强大的 IDE，已支持 TypeScript 开发，收费。

3.10.4 展望未来

提到 TypeScript，很容易跟另一个著名的 JavaScript 转译型语言 CoffeeScript 做对比。曾经的 CoffeeScript 也是风靡前端的开发语言，而随着 ES 6 的崛起，CoffeeScript 已经不再流行，在前端社区中诞生了各种 **CoffeeScript to ES 6** 的工具。

TypeScript 会不会成为下一个 CoffeeScript 呢？这也许是一直萦绕在 TypeScript 开发者心里的一个困扰。事实上，TypeScript 积极地拥抱 ECMAScript 标准，并在此基础上增加了大量优秀的新特性；另外，有微软这个大厂商背书，加上繁盛的 TypeScript 生态圈，可以预见 TypeScript 未来的繁荣。

3.11 小结

本章首先介绍了 TypeScript 的背景，接着讲述了 TypeScript 的主要特性，包括基本类型、函数、类、接口、装饰器、模块和泛型等内容；最后介绍了 TypeScript 周边的其他知识点，包括编译配置、开发工具、声明文件及未来展望。

通过本章的学习，我们基本上掌握了 TypeScript 的相关知识点，具备使用 TypeScript 开发 Angular 应用的能力了。在下一章中，将通过对两个简单小项目的讲解，让我们快速了解 Angular 应用是如何搭建的。

4

快速入门

通过前面章节的介绍，相信大家对 TypeScript 与 Angular 已经有了初步的了解。在接下来的内容中，开始介绍 Angular 环境搭建的一些准备工作，包括安装 Node.js、npm 等；然后介绍如何使用 Angular CLI 搭建 Angular 的开发环境，当环境搭建完成后就可以在此基础上进行开发了；接下来，在所搭建的环境中通过编写一个简单的 Hello World 例子来开始第一个 Angular 应用程序的开发；再利用一些命令行工具将整个项目运行起来；最后进一步介绍通讯录例子的其他内容。

4.1 Hello World 例子

在本节中，我们将介绍如何使用 Angular CLI 搭建环境和运行 Hello World 示例。

4.1.1 准备工作

安装 NodeJS

在开始搭建环境前，还需要做一些准备工作，包括安装 Angular 所依赖的基础环境 Node.js，可以在官网（<https://nodejs.org/en/download/>）下载安装，本例中使用的 Node.js 版本是 7.9.0。安装完成后，可以在命令行窗口中输入以下命令来查看 Node.js 的版本：

```
$ node -v
```

Node.js 安装包集成了 npm，可以通过以下方式来看 npm 的版本：

```
$ npm -v
```

安装 Angular CLI

在安装 Angular CLI 之前，请确认系统中的 Node.js 版本高于 6.9.0 且 npm 版本高于 3.0.0。

安装 Angular CLI 只需要在命令行窗口中输入如下命令即可：

```
$ npm install -g @angular/cli@1.5.0
```



本例固定使用 1.5.0 版本，若需要使用 npm 最新版本，可删除 @1.5.0。

安装完成后，通过如下命令查看 Angular CLI 的安装版本。

```
ng -v
```

或者

```
ng version
```

```
@angular/cli: 1.5.0
```

```
node: 8.8.1
```

```
os: darwin x64
```

4.1.2 构建项目

确定 Angular CLI 已经安装成功后，可以开始创建 Hello World 项目了。

创建项目

执行如下命令创建 Hello World 项目：

```
$ ng new angular-hello-world --skip-install
```



ng new 命令在创建完项目后，默认会执行 npm install 安装依赖包，添加 --skip-install 参数可跳过安装过程。关于依赖包的安装下一步继续介绍。

命令执行完成后，大致输出如下：

```
create angular-hello-world/README.md (1033 bytes)
create angular-hello-world/.angular-cli.json (1254 bytes)
create angular-hello-world/.editorconfig (245 bytes)
create angular-hello-world/.gitignore (516 bytes)
create angular-hello-world/src/assets/.gitkeep (0 bytes)
create angular-hello-world/src/environments/environment.prod.ts (51 bytes)
create angular-hello-world/src/environments/environment.ts (387 bytes)
create angular-hello-world/src/favicon.ico (5430 bytes)
create angular-hello-world/src/index.html (304 bytes)
create angular-hello-world/src/main.ts (370 bytes)
create angular-hello-world/src/polyfills.ts (2667 bytes)
create angular-hello-world/src/styles.css (80 bytes)
create angular-hello-world/src/test.ts (1085 bytes)
create angular-hello-world/src/tsconfig.app.json (211 bytes)
create angular-hello-world/src/tsconfig.spec.json (304 bytes)
create angular-hello-world/src/typings.d.ts (104 bytes)
create angular-hello-world/e2e/app.e2e-spec.ts (301 bytes)
create angular-hello-world/e2e/app.po.ts (208 bytes)
create angular-hello-world/e2e/tsconfig.e2e.json (235 bytes)
create angular-hello-world/karma.conf.js (923 bytes)
create angular-hello-world/package.json (1324 bytes)
create angular-hello-world/protractor.conf.js (722 bytes)
create angular-hello-world/tsconfig.json (363 bytes)
create angular-hello-world/tslint.json (2985 bytes)
create angular-hello-world/src/app/app.module.ts (316 bytes)
create angular-hello-world/src/app/app.component.css (0 bytes)
create angular-hello-world/src/app/app.component.html (1120 bytes)
create angular-hello-world/src/app/app.component.spec.ts (986 bytes)
create angular-hello-world/src/app/app.component.ts (207 bytes)
Successfully initialized git.
Project 'angular-hello-world' successfully created.
```

看到有以上信息的输出，说明项目创建成功。

项目结构

切换到项目 `angular-hello-world` 目录，文件结构如下：


```
.
├── README.md
├── e2e
│   ├── app.e2e-spec.ts
│   ├── app.po.ts
│   └── tsconfig.e2e.json
├── karma.conf.js
├── package.json
├── protractor.conf.js
├── src
│   ├── app
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.spec.ts
│   │   ├── app.component.ts
│   │   └── app.module.ts
│   ├── assets
│   ├── environments
│   │   ├── environment.prod.ts
│   │   └── environment.ts
│   ├── favicon.ico
│   ├── index.html
│   ├── main.ts
│   ├── polyfills.ts
│   ├── styles.css
│   ├── test.ts
│   ├── tsconfig.app.json
│   ├── tsconfig.spec.json
│   └── typings.d.ts
├── tsconfig.json
└── tslint.json
```

下面简单介绍项目根目录下的一些重要文件夹及文件。

- **e2e**: 项目测试文件的存放位置。
- **src/app**: 项目源码存放的位置。
- **src/assets**: 项目静态资源文件的存放位置, 如图片文件、样式文件、脚本文件等。
- **src/environments**: 存放定义应用程序行为的文件, 如开发环境、生产环境、测试环境等。

- `main.ts`: 项目入口文件。
- `polyfills.ts`: 平台 API 兼容文件, 如需要兼容 IE 9+、Firefox 浏览器, 则需要配置此文件。
- `styles.css`: 项目全局样式文件, 作用于整个项目。
- `typings.d.ts`: TypeScript 配置外部变量的文件, 如 JQuery。
- `tsconfig.json`: TypeScript 行为配置文件, 可以定义 TypeScript 的默认行为。

运行项目

安装项目依赖包:

```
$ npm install
```



在创建项目时, 如果使用 `--skip-install` 跳过了默认的依赖包安装步骤, 就需要手动执行 `npm install`。这里也可以使用其他更快的 `npm` 仓库源进行安装, 如淘宝 NPM 镜像 (<https://npm.taobao.org/>), 增加依赖包安装的成功率。

然后, 启动应用:

```
$ ng serve
```

服务成功启动后, 会看到类似于如下的输出信息:

```
** NG Live Development Server is listening on localhost:4200, open your browser on  
http://localhost:4200/ **
```

```
Date: xxx
```

```
Hash: 36d3ae05761b1fdc1173
```

```
Time: 7669ms
```

```
chunk {inline} inline.bundle.js (inline) 5.79 kB [entry] [rendered]
```

```
chunk {main} main.bundle.js (main) 20.7 kB [initial] [rendered]
```

```
chunk {polyfills} polyfills.bundle.js (polyfills) 553 kB [initial] [rendered]
```

```
chunk {styles} styles.bundle.js (styles) 33.8 kB [initial] [rendered]
```

```
chunk {vendor} vendor.bundle.js (vendor) 7.03 MB [initial] [rendered]
```

```
webpack: Compiled successfully.
```

最后, 在浏览器中输入 `localhost:4200` (默认端口号为 4200), 最终看到的效果如图 4-1 所示。

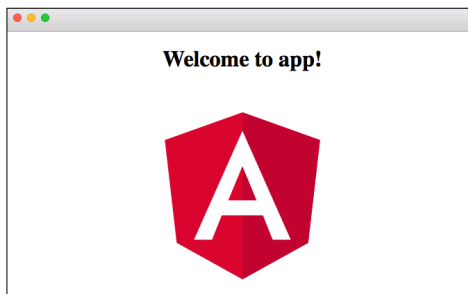


图 4-1 效果图

编辑第一个组件

修改 src/app/app.component.ts 文件为如下内容：

```
// app.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  welcome = 'Hello, Angular';
}
```

同时，修改 app.component.html 文件为如下内容：

```
<!-- app.component.html -->
<h1 style="font-size:50px;">
  {{ welcome }}!
</h1>
```

保存修改，打开浏览器查看修改后的效果，如图 4-2 所示。

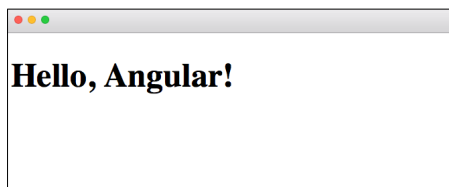


图 4-2 修改后效果图

4.2 通讯录例子

4.2.1 背景介绍

通过上一节的 Hello World 项目例子，相信大家对开发 Angular 应用的步骤已经有了较为清晰的了解。接下来我们将使用 Angular 搭建一个简单的通讯录例子，并实现一些功能，如获取和展示联系人名单、编辑联系人、添加联系人及收藏联系人等。

读者通过预览图可以了解整个 App 的情况，首先展示的是联系人列表页面，如图 4-3 所示。

联系人列表页面会显示所有的联系人，在每个联系人信息中会显示出联系人的头像、姓名与电话号码。单击每个联系人，都会跳转到详情页面，显示该联系人更详细的信息。在页面的右上角有一个“添加”按钮，单击会跳转到添加联系人页面。在页面的底部有一个导航控件，可以实现联系人列表页面与联系人收藏页面的相互切换。联系人收藏页面如图 4-4 所示。

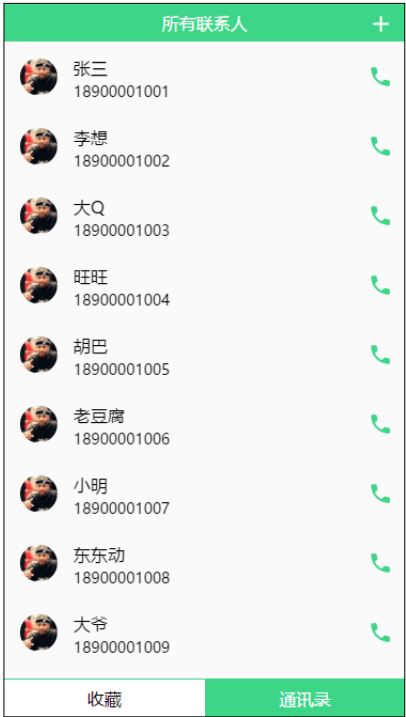


图 4-3 联系人列表页面



图 4-4 联系人收藏页面

所收藏的联系人都会通过列表的形式显示在该页面上，并且单击每个联系人，都会跳转到联系人详情页面，如图 4-5 所示。

在联系人详情页面中，最上面会显示联系人的头像、姓名，以及一个星形的收藏按钮，单击这个按钮就会收藏或者取消收藏该联系人，同时所收藏的联系人将会在收藏页面显示出来。在联系人详情页面的下方是联系人的详细信息表单，包括电话、生日、住址等。这些信息都是可以编辑的，单击页面右上角的“编辑”按钮，就会跳转到编辑页面，如图 4-6 所示。



图 4-5 联系人详情页面



图 4-6 编辑页面

编辑页面是编辑与添加联系人功能共用的页面。当处于编辑状态时，会显示已经填写过的联系人信息，用户可以修改这些信息；当处于添加状态时，表单是空白的，用户可以填写并提交内容。如果修改或者添加的内容格式不对，当输入框失去焦点时，就会在右边显示一条红色的竖线作为提示，格式正确则显示绿色的竖线。

上面介绍了每个页面的功能，这些页面之间的交互关系如图 4-7 所示。

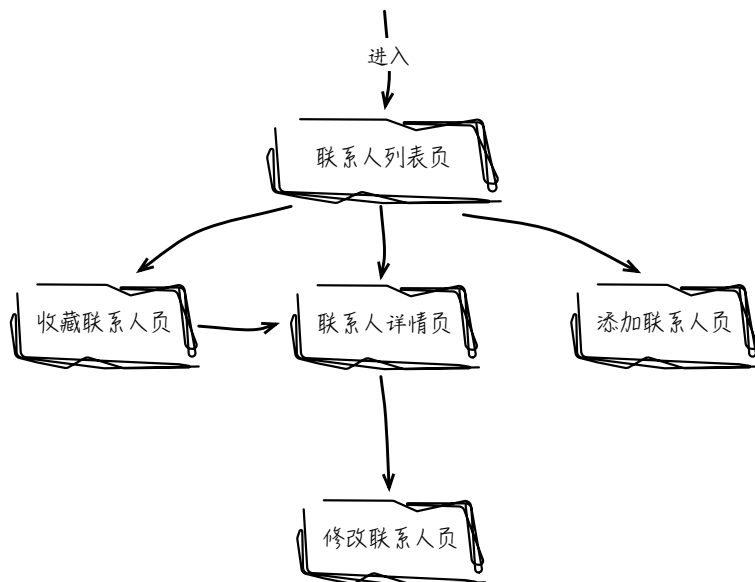


图 4-7 通讯录各页面交互关系图

4.2.2 架构设计

通过上面的介绍，我们了解了通讯录的功能与交互，接下来就在搭建好的 Hello World 例子的基础上开始通讯录项目实战。

这里先列出通讯录项目的主要目录结构，如下所示。

```
.
├── LICENSE
├── README.md
├── package.json
├── src
│   ├── app
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.ts
│   │   ├── app.module.ts
│   │   ├── app-routing.module.ts
│   │   └── collection
│   │       ├── collection.component.css
│   │       ├── collection.component.html
│   │       └── collection.component.ts
```

```

|   |   |   └── index.ts
|   |   └── detail
|   |       ├── detail.component.css
|   |       ├── detail.component.html
|   |       ├── detail.component.ts
|   |       └── index.ts
|   └── edit
|       ├── edit.component.css
|       ├── edit.component.html
|       ├── edit.component.ts
|       └── index.ts
|   └── list
|       ├── index.ts
|       ├── item.component.css
|       ├── item.component.html
|       ├── item.component.ts
|       ├── list.component.css
|       ├── list.component.html
|       └── list.component.ts
|   └── shared
|       ├── btn-click.directive.ts
|       ├── contact.service.ts
|       ├── footer.component.css
|       ├── footer.component.html
|       ├── footer.component.ts
|       ├── header.component.css
|       ├── header.component.html
|       ├── header.component.ts
|       ├── index.ts
|       ├── phone.pipe.ts
|       └── util.service.ts
|   ├── assets
|   ├── index.html
|   ├── main.ts
|   ├── polyfills.ts
|   └── styles.css
└── .angular-cli.json
└── tsconfig.json
└── tslint.json

```

在通讯录项目中，主要分为四大模块：联系人列表模块（list）、联系人详情模块（detail）、编辑模块（edit）及收藏模块（collection）。在这四个模块中，页面的跳转交互是怎么实现的呢？Angular 提供了路由模块，用来完成页面间的跳转。在该项目中，创建了一个单独的文件，位于 `src/app/app-routing.module.ts`，配置了该项目的路由，代码如下：

```
// src/app/app-routing.module.ts
// 省略 import
export const routes: Routes = [
  {
    path: '',
    redirectTo: 'list',
    pathMatch: 'full'
  },
  {
    path: 'list',
    component: ListComponent
  },
  {
    path: 'list/:id',
    component: DetailComponent
  },
  {
    path: 'edit',
    component: EditComponent
  },
  {
    path: 'edit/:id',
    component: EditComponent
  },
  {
    path: 'collection',
    component: CollectionComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
```



```
  })  
  export class AppRoutingModule { }
```

配置好的路由信息可以在 `app.module.ts` 文件中引入使用。关于路由的介绍可以在后续章节中深入学习。最终入口文件的代码如下：

```
// app.module.ts  
// 省略 import  
@NgModule({  
  declarations: [  
    AppComponent,  
    ListComponent,  
    ListItemComponent,  
    DetailComponent,  
    CollectionComponent,  
    EditComponent,  
    HeaderComponent,  
    FooterComponent,  
    PhonePipe,  
    BtnClickDirective  
  ],  
  imports: [  
    BrowserModule,  
    FormsModule,  
    HttpClientModule,  
    AppRoutingModule  
  ],  
  providers: [ContactService, UtilService],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```

项目的主要模块都会被引入到 `app.module.ts` 文件中。

在项目开发中，数据操作是很重要的部分。Angular 对数据的增、删、改、查是通过特定的服务实现的，并将这些服务注入到 `NgModule` 中，这样在 `NgModule` 中引入的组件就可以直接调用其中的方法，从而达到数据交互的目的。服务的基本写法如下：

```
import { Injectable } from '@angular/core';  
import { HttpClient } from '@angular/common/http';
```

```
@Injectable()
export class ContactService {
  constructor(
    private http: HttpClient
  ) {}

  // ...
}
```

在上述代码中, `@Injectable()` 表示 `ContactService` 需要注入它所依赖的其他服务 (如 HTTP 服务)。关于服务与依赖注入, 在后续章节中会着重介绍, 这里不再赘述。

上面的内容介绍了通讯录示例的主要基础模块, 包括页面跳转所用到的路由配置及数据服务。下面再通过两张图来总览整个通讯录示例的技术点与内容, 如图 4-8、图 4-9 所示。

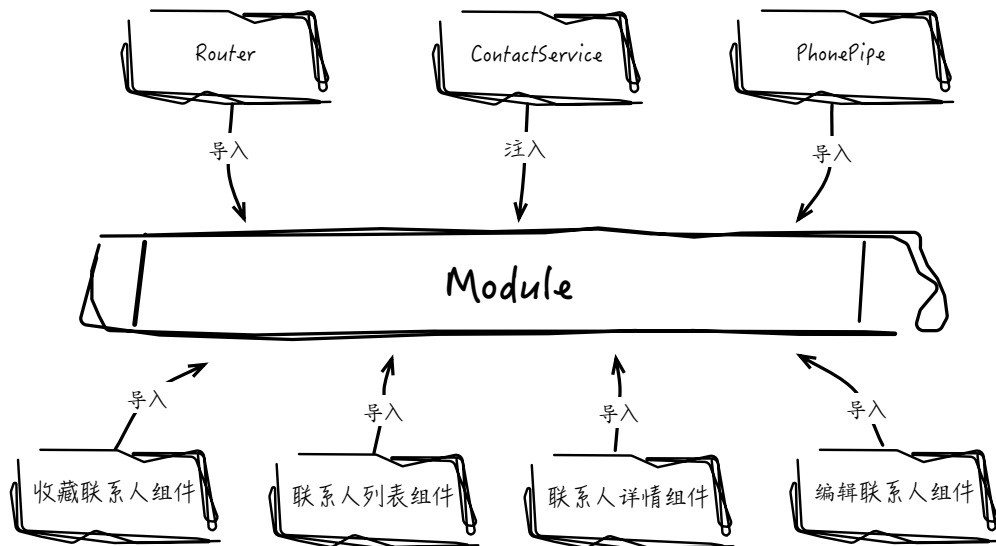


图 4-8 NgModule 组成图

图 4-8 主要介绍了在通讯录示例中, 将所涉及的组件、路由、服务和管道等引入到 NgModule 中, 并组成一个整体可以运行起来的大模块。

图 4-9 则详细地剖析了 NgModule 里面的各部分之间的关系与运行机制。

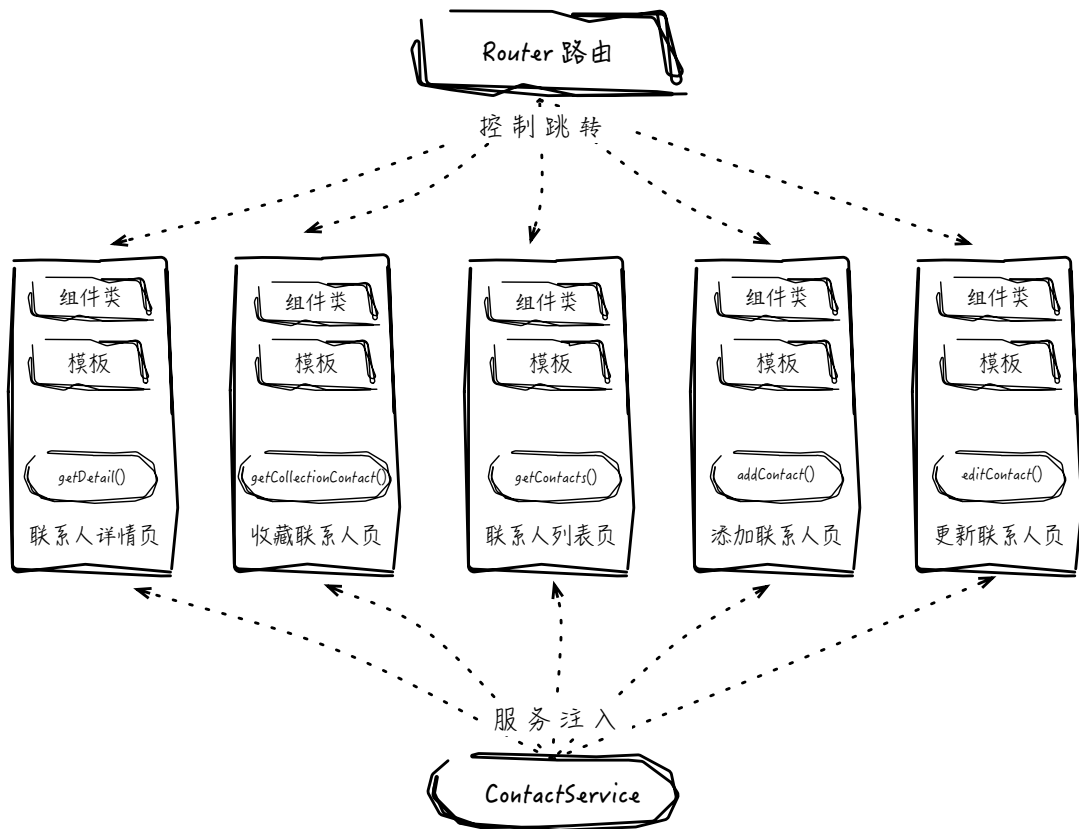


图 4-9 NgModule 详细图

通过这两张图，可以了解通讯录示例所用到的相关 Angular 知识点。至此，我们就介绍完了通讯录示例的整体结构，对 Angular 应用开发的过程也有了较为直观的感受。接下来需要针对每个组件模块编写对应的业务逻辑，这里就不具体展开讲解了，读者可以到 GitHub 上查看完整的代码。



本章所介绍的 Hello World 及通讯录例子源码都可以在 GitHub 下载。

- Hello World 例子的下载地址：<https://github.com/angular-programming/angular-hello-world>;
- 通讯录例子的下载地址：<https://github.com/angular-programming/angular-contacts-demo>。

4.3 小结

在本章中，通过 Hello World 例子，我们主要基于 Angular CLI 工具搭建了 Angular 基础开发环境；随后通过通讯录项目，进一步了解了 Angular 所包含的一些知识点，包括 NgModule、路由、组件、服务等。在接下来的第二部分的章节中，我们将会具体介绍这些知识点，使读者深入了解并最终掌握 Angular 应用的开发。

第二部分

深入篇

- Angular 架构总览
- 组件
- 模板
- 指令
- 服务与 RxJS
- 依赖注入
- 路由
- 测试

5 Angular 架构总览

上述章节带领读者学习了 Angular 相关知识,包括其历史发展、周边生态、TypeScript 语法,并通过例子快速上手了 Angular 应用的开发。接下来本书的第二部分会将重心回归到 Angular 框架本身,开始深入揭开 Angular 的技术内幕。

本章首先从总览的角度来分析 Angular 的各个核心组成部分,让读者对框架有一个整体的认知,然后对 Angular 项目源码模块构造进行简要分析。

Angular 彻底重写了 AngularJS 1.x,这多少会给社区带来不便,好在 Angular 团队在无缝升级方面下了不少功夫,而且更重要的是,重写能让 Angular 抛掉老版的包袱。采用新架构设计的 Angular 代码更加简洁、易读,性能更好,更加贴合新时代前端的发展趋势,例如基于组件的设计、响应式编程等。除此之外,Angular 的适用场景更广,例如支持服务端渲染,能更好地适配 Mobile 应用 (Mobile Toolkit),以及支持离线编译等。

5.1 核心模块介绍

一个完整的 Angular 应用主要由 6 个重要部分构成,分别是:组件、模板、指令、服务、依赖注入和路由。这些组成部分各司其职,而又紧密协作,它们的关系如图 5-1 所示。

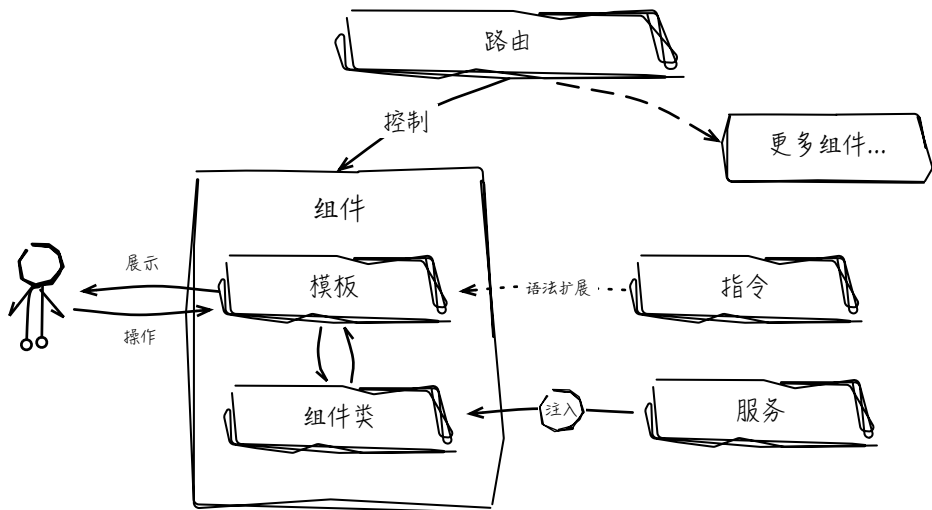


图 5-1 Angular 核心模块关联图

与用户直接交互的是模板视图，模板视图并不是独立的模块，它是组成组件的要素之一。另一个要素是组件类，用以维护组件的数据模型及功能逻辑。路由的功能是控制组件的创建和销毁，从而驱使应用界面跳转切换。指令与模板相互关联，最重要的作用是增强模板特性，间接扩展了模板的语法。服务是封装若干功能逻辑的单元，这个功能逻辑可以通过依赖注入机制引入到组件内部，作为组件功能的扩展。

在 Angular 的应用接收用户指令、加工处理后输出相应视图的过程中，组件始终处于这个交互的出入口，这正是 Angular 基于组件设计的体现。组件承载着 Angular 的核心功能，所以接下来的内容将从组件开始，逐步揭开 Angular 框架的神秘面纱。

5.1.1 组件

Angular 框架基于组件设计，其应用由一系列大大小小松耦合的组件构成，那么组件到底意味着什么？下面将通过通讯录来进行说明，其演示效果如图 5-2 所示。

实际上，所有框起来的部分均是由相应的组件渲染的，并且这些组件层层嵌套、自上而下构成组件树。例如最外层的方框为根组件，包含了 Header、ContactList 及 Footer 三个子组件，其中 ContactList 又有自己的子组件，如图 5-3 所示。

树状结构的组件关系意味着每个组件并不是孤立存在的，父子组件之间存在着双向的数据流动。要理解数据是怎样流动的，首先要了解组件之间的调用方式。简单地说，组件的外在形态就是自定义标签，所以组件的调用实际体现在模板标签里的引用上。Contact 组件的示例代码如下：



图 5-2 组件形态

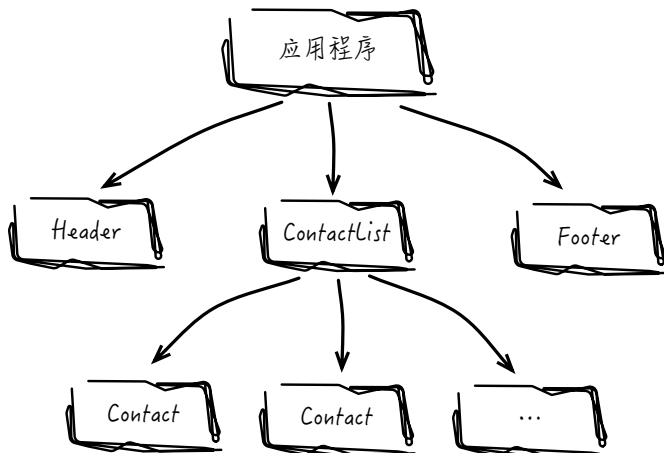


图 5-3 应用的组件树

```

@Component({
  selector: 'contact',
  template: '<div>xxx</div>' // 省略部分内容
})
export class ContactComponent {
  @Input() item: ContactModel;
  @Output() update: EventEmitter<ContactModel>;
  constructor() {}
  // ...
}

```

`@Input()` 和 `@Output` 声明了组件 `Contact` 对外暴露的接口，`item` 变量用来接收来自父组件的数据源输入，`update` 接口用于向父组件发送数据。那么父组件是如何引用子组件并调用这些接口的呢？`ContactList` 父组件的示例代码如下：

```

@Component({
  selector: 'contact-list',
  template: `
    <!-- ... -->
    <!--使用<contact>标签调用ContactComponent组件-->
    <contact [item]="listItem[0]" (update)="doUpdate(newItem)"></contact>
    <!-- ... -->
  `
})

```



```
\n\n    })\n  export class ContactListComponent {\n    listItem: ContactModel[];\n    constructor() {}\n    doUpdate(item: ContactModel) {\n      // ... \n    }\n  }\n}
```



在父组件 `ContactListComponent` 的模板里直接使用子组件 `ContactComponent` 定义的标签，需要依赖“模块”的特性。关于“模块”的内容在第 6 章中会进行介绍。

由 `template` 属性值可见，父子组件之间通过类似于 HTML 属性的方式传递数据，其中 `[item]` 称为属性绑定，数据从父组件流向子组件；`(update)` 称为事件绑定，数据从子组件流向父组件，如图 5-4 所示。

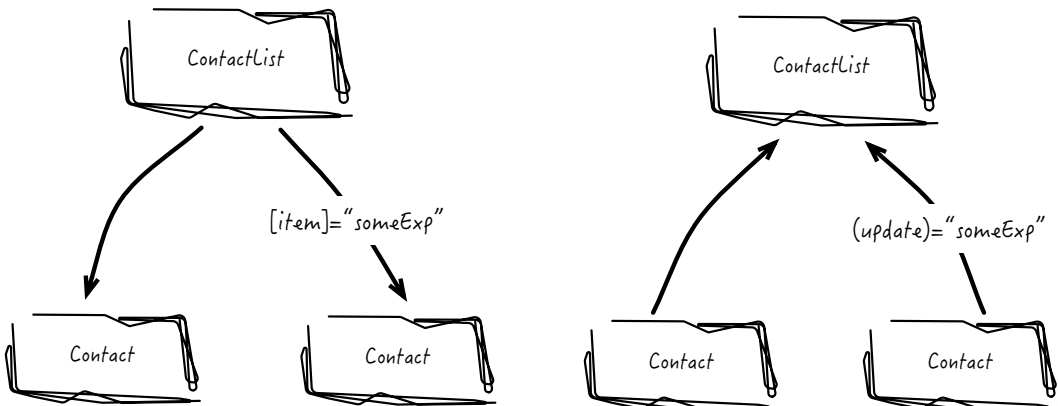


图 5-4 父子组件之间的数据流动

细心的读者可能已经发现，在 Angular 的模板里可以直接引用组件的成员属性，如 `listItem` 和 `doUpdate`。组件类和模板之间的数据交互称为数据绑定，前面所说的属性绑定和事件绑定也属于数据绑定的范畴，属性绑定和事件绑定既可用于父子组件之间的数据传递，也可用于组件数据模型和模板视图之间的数据传递。所以在父子组件通信的过程中，模板充当桥梁的角色，连接着两者的功能逻辑。

如图 5-5 所示，这就是 Angular 的数据流动机制。然而，流动并不是自发形成的，流动需要一个驱动力，这个驱动力即 Angular 的变化监测机制。Angular 是一个响应式系统，每次数据变动几乎都能实时处理，并更新对应的视图。那么 Angular 是如何感知数据对象发生了变动呢？ES 5 提供了 `getter/setter` 语言接口来捕获对象变动，然而 Angular 并没有采用。Angular 是在适当的时机中检验对象的值是否被改动的，这个适当的时机并不是指固定的某个频率，而通常是在用户操作事件（如单击）或者 `setTimeout`、XHR 回调等这些异步事件触发之后。Angular 捕获这些异步事件的工作是通过 `Zone.js` 库实现的（关于 `Zones` 的内容在第 6 章会展开讲述）。

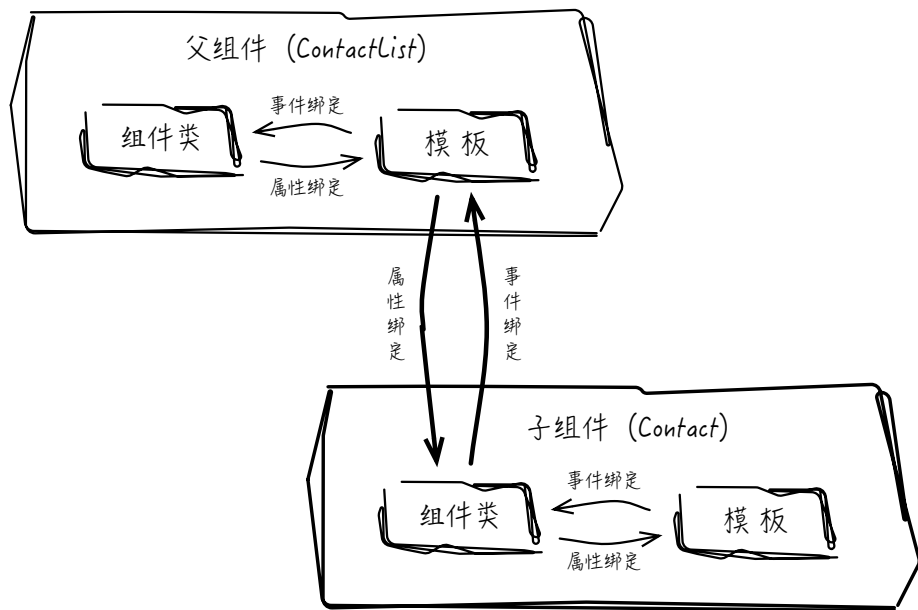


图 5-5 数据流动机制

如图 5-6 所示，每个组件背后都维护着一个独立的变化监测器，这个变化监测器记录着所属组件的数据变更状态。由于应用是以组件树的形式组织的，因此每个应用也都对应着一棵变化监测树。当 `Zones` 捕获到某异步事件后，它会通知 Angular 执行变化监测操作，每次变化监测操作都始于根组件，并以深度优先的原则向叶子组件遍历执行。



Angular 5.0 提供了不依赖 `Zones` 的写法，更多的详情在第 6 章中讲述。

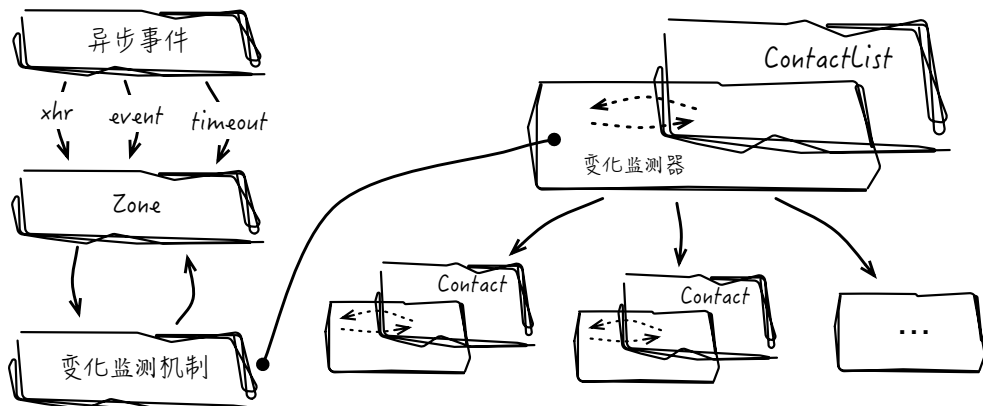


图 5-6 变化监测机制

Angular 强大的数据变化监测机制使得开发者不必关心数据何时变动，结合数据绑定实现模板视图实时更新。变化监测机制提供了数据自动更新功能，若此时需要手动捕获变化事件做一些额外处理，可以吗？答案是肯定的。Angular 还提供了完善的生命周期钩子给开发者调用，如 `ngOnChanges` 可以满足刚提到的捕获变化事件的要求，`ngOnDestroy` 可以在组件销毁前做一些清理工作，等等。

以上是关于组件的简述，组件在 Angular 框架中处于最核心的位置，更多的关于组件的内容会在第 6 章中继续剖析。

5.1.2 模板

Angular 模板基于 HTML，普通的 HTML 也可作为模板输入，示例代码如下：

```
@Component({
  selector: 'contact',
  template: `
    <div>
      <span> 张三 </span>
    </div>
  `,
})
export class ContactComponent {}
```

但 Angular 模板不止如此，Angular 还为模板定制了一套强大的语法体系，涉及内容颇多，这也是为什么将模板单独列出的原因。数据绑定是模板最基本的功能，除了上述提到的属性绑定和事件绑定，插值也是很常见的数据绑定语法，示例代码如下：

```

@Component({
  selector: 'contact',
  template: `
    <div>
      <span>{{ item.name }}</span>
    </div>
  `,
})
export class ContactComponent {
  @Input() item: ContactModel;
  // ...
}

```

插值语法是由一对双大括号“{{}}”组成的，插值的变量上下文是组件类本身，如上例中的 `item`，插值是一种单向的数据流动——从数据模型到模板视图。

上面提到的三种数据绑定（即属性绑定、事件绑定及插值）语法的数据流动都是单向的，但是在某些场景下需要双向的数据流动支持（如表单）。结合属性绑定和事件绑定，Angular 模板可实现双向绑定的功能。例如：

```
<input [(ngModel)]="contact.name"></input>
```

`[()]` 是实现双向绑定的语法糖，`ngModel` 是辅助实现双向绑定的内置指令。上述代码执行后，`Input` 控件和 `contact.name` 之间就形成双向的数据关联，`Input` 的值发生变化时，可自动赋值至 `contact.name`，而 `contact.name` 的值被组件类改变时，也可实时更新 `Input` 的值。更多的关于双向绑定的实现细节，在第 7 章中会详细展开。

由上可知，数据绑定负责数据的传递与展示，而针对数据的格式化显示，Angular 提供了一种称作管道的功能，使用竖线“|”来表示，示例代码如下：

```
<span>{{ contact.telephone | phone }}</span>
```

假设上述 `contact.telephone` 的值是 18612345678，这一串数字并不太直观，管道命令 `phone` 可以将其进行美化输出，如“186-1234-5678”，而不影响 `contact.name` 本身的值。管道支持开发者定制开发，`phone` 即属于自定义管道。Angular 也提供了一些基本的内置管道命令，如格式化数字的 `number`、格式化日期的 `date` 等。

Angular 模板还有很多强大的语法特性，包括上面提到的组件所封装的自定义标签，如 `<contact></contact>`。此外，还提供了一套强大的“指令”机制来简化一些特定的交互场景，如样式处理、数据遍历及表单处理等，这些强大的语法特性将会在第 7 章中一一罗列。

5.1.3 指令

指令与模板关系密切，指令可以与 DOM 进行灵活交互，它或者改变样式，或者改变布局。指令的范畴很广，实际上组件也是指令的一种。组件与一般指令的区别在于：组件带有单独的模板，即 DOM 元素；而一般的指令作用在已有的 DOM 元素上。一般的指令分为两种：结构指令和属性指令。

结构指令能够添加、修改或删除 DOM，从而改变布局，如 `ngIf`：

```
<button *ngIf="canEdit"> 编辑 </button>
```

当 `canEdit` 的值为 `true` 时，`button` 按钮会显示到视图上；当 `canEdit` 的值为 `false` 时，`button` 按钮会从 DOM 树上移除。



注意 * 不能丢掉，这是语法的重要部分。

属性指令用来改变元素的外观或行为，使用起来跟普通的 HTML 元素属性非常相似，例如 `ngStyle` 指令，用于动态计算样式值。示例代码如下：

```
<span [ngStyle]="setStyles()">{{ contact.name }}</span>
```

`` 标签的样式由 `setStyles()` 函数计算得出，`setStyles()` 是其组件类的成员函数，返回一个计算好的样式对象。示例代码如下：

```
class ContactComponent {  
  
  private isImportant: boolean;  
  
  setStyles() {  
    return {  
      'font-size': '14px',  
      'font-weight': this.isImportant ? 'bold' : 'normal'  
    }  
  }  
}
```

上面列举的 `ngIf` 和 `ngStyle` 都是 Angular 的内置指令，类似的还有 `ngFor`、`ngClass` 等。这些内置指令的作用更偏向于为模板提供语法支持，所以在第 7 章中会连带讲述内置指令的用法。指令更具吸引力的地方在于支持开发者自定义，自定义指令能最大限度地实现 UI 层面的逻辑复用。关于详细的构建自定义指令的过程将会在第 8 章中讲述。

5.1.4 服务

服务是封装单一功能的单元，类似于工具库，常被引用到组件内部，作为组件的功能扩展。那么服务包含什么？它可以是一个简单的字符串或 JSON 数据，也可以是一个函数，甚至是一个类，几乎所有的对象都可以封装成服务。以日志服务为例，一个简单的日志服务如下：

```
export class LoggerService {  
  info(msg: any) { console.log(msg); }  
  warn(msg: any) { console.warn(msg); }  
  error(msg: any) { console.error(msg); }  
}
```

组件记录日志时只需注入 `LoggerService` 服务即可调用其接口，封装成独立模块的日志服务使其能被所有的组件所复用，这就是服务设计的原则。

HTTP 是 Angular 里常用的内置服务，它封装了一系列的异步数据请求接口，但与一般的接口不同，HTTP 服务对外暴露的是 Reactive Programming 规范的接口，基于 RxJS 实现，严格贯彻响应式编程思想。所以在第 9 章除介绍 HTTP 服务的使用方法之外，还会有专门的章节讲解 RxJS 这个流行的响应式编程框架。

5.1.5 依赖注入

在服务章节中会提到“注入”这个概念，依赖注入一直都是 Angular 的卖点。通过依赖注入机制，服务等模块可以被引入到任何一个组件（或模块，或其他服务）中，而开发者无须关心这些模块是如何被初始化的。因为 Angular 已经帮助处理好，包括该模块本身依赖的其他模块也会被初始化。如图 5-7 所示，当组件注入日志服务后，日志服务，以及它所依赖的基础服务都会被初始化。

可以说，依赖注入是一种帮助开发者管理模块依赖的设计模式。在 Angular 中，依赖注入与 TypeScript 相结合提供了更好的开发体验。在 TypeScript 中，对象通常被明确赋予类型，通过类型匹配，组件类便可知道该用哪种类型实例来赋值变量。一个简单的依赖注入示例如下：

```
import { LoggerService } from './logger-service';  
  
@Component({  
  selector: 'contact-list',  
  providers: [LoggerService]  
})
```

```
export class ContactListComponent {  
  constructor(private logger: LoggerService) {}  
  
  doSomething() {  
    this.logger.info('xxx');  
  }  
}
```

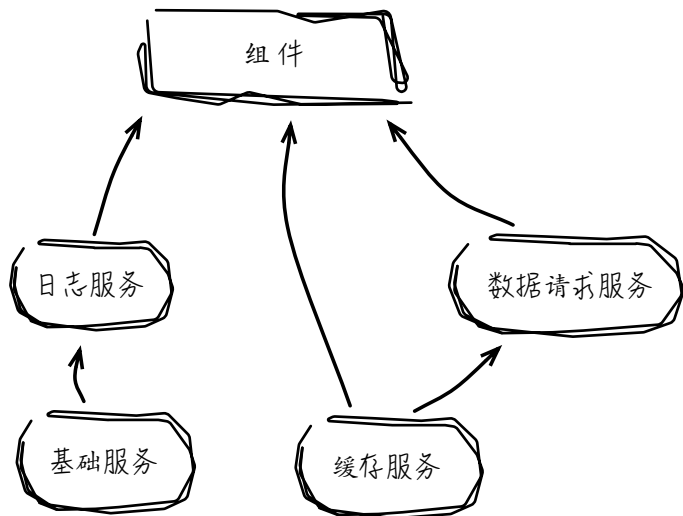


图 5-7 依赖注入



`private logger: LoggerService` 创建了一个类的私有属性，详细信息可参阅第 3 章。

`@Component` 装饰器中的 `providers` 属性是依赖注入操作的关键，它会为该组件创建一个注入器对象，并新建 `LoggerService` 实例存储到这个注入器里。组件在需要引入 `LoggerService` 实例时，通过 TypeScript 的类型匹配即可从注入器中取出相应的实例对象，无须再重复显式实例化。

需要注意的是，服务的每一次注入（也就是使用 `providers` 声明），该服务都会被创建出新的实例，组件的所有子组件均默认继承父组件的注入器对象，复用该注入器里存储的服务实例。这种机制可保证服务以单例模式运行，除非某个子组件再次注入（即通过 `providers` 声明），如图 5-8 所示。

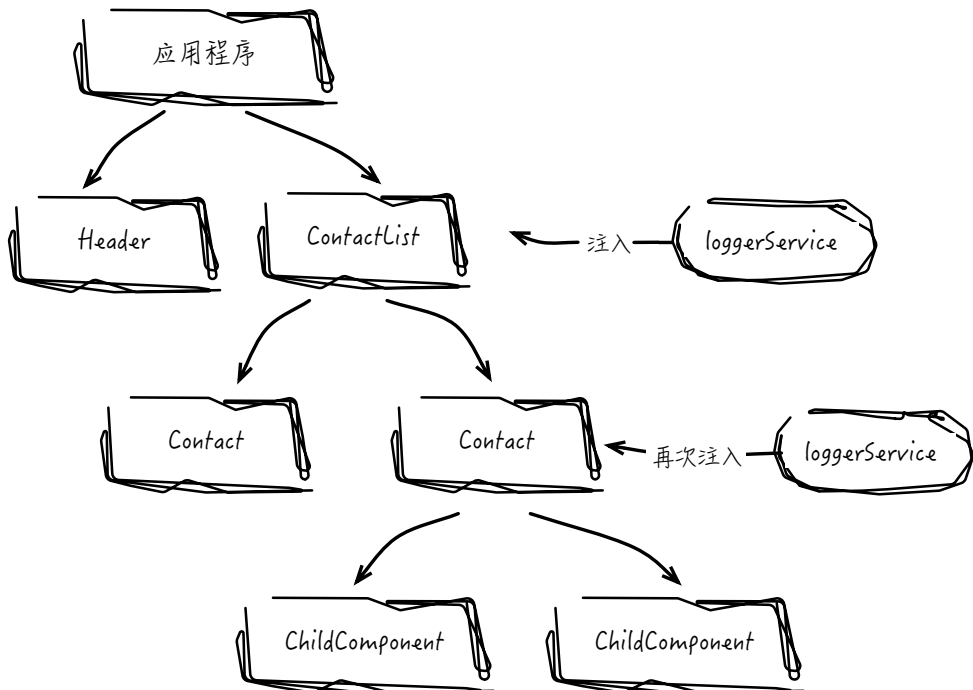


图 5-8 注入机制

在 `ContactList` 组件注入的 `LoggerService` 是可以被 `ContactList` 及其子组件使用的。不过，当子组件 `Contact` 又重新注入了新的 `LoggerService` 后，`Contact` 及其子组件使用的将会是新创建的 `LoggerService` 服务实例。这种灵活的注入方式可以适应多变的应用情景，既可配置全局的单例服务（在应用的根组件注入即可），也可按需注入不同层级的服务，彼此的数据状态不会相互影响。更多的关于注入的细节会在第 10 章中继续讲解。

5.1.6 路由

Angular 作为一个单页应用框架，前端路由功能必不可少。在 Angular 中，路由的作用是建立 URL 路径和组件之间的对应关系，根据不同的 URL 路径匹配出相应的组件并渲染。假设通讯录应用需要添加一个通话记录页面，简单的路由配置如下：

```
[
  {path: '', component: ContactListComponent},
  {path: 'record', component: RecordListComponent},
  // ...
]
```


注意该配置的第一项 path 的值为空，这表示默认路由。上述配置的作用如下：

- 访问 `http://www.abc.com/` 时，页面渲染 `ContactListComponent` 组件。
- 访问 `http://www.abc.com/record` 时，页面渲染 `RecordListComponent` 组件。

组件树的节点会不断发生变化，如图 5-9 所示。

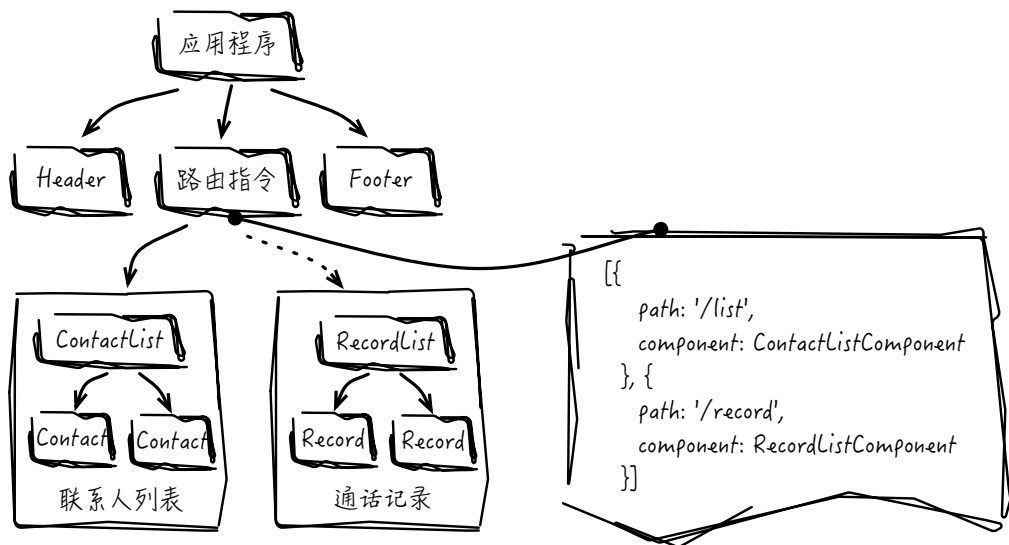


图 5-9 基本路由功能

原来的组件树中多了一个路由指令（标签名为 `<router-outlet>`），图 5-9 中应用程序的模板如下：

```
<div>
  <header></header>
  <router-outlet></router-outlet>
  <footer></footer>
</div>
```

路由指令 `router-outlet` 起着类似于“插座”的作用，根据当前的 URL 路径匹配插入对应的组件节点，实现了主体内容（页面）的刷新，这就是 Angular 路由最基本的功能。路由指令还支持多重嵌套，实现子路由功能。假设通讯录应用的通话记录页面需要新增标签页切换功能，用来切换显示全部来电及未接来电，可以修改路由配置如下：

```
[
  {path: 'list', component: ContactListComponent},
  { path: 'record', component: RecordListComponent, children: [
```

```
{path: '', component: AllRecordsComponent},  
{path: 'miss', component: MissRecordsComponent}  
]  
},  
// ...  
]
```

上面配置中的 record 条目新增了一个 children 的配置项，用于设置子路由的信息。当 URL 改变为 <http://www.abc.com/record> 时，显示的是 AllRecordsComponent 组件视图，通话记录组件子路由功能如图 5-10 所示。

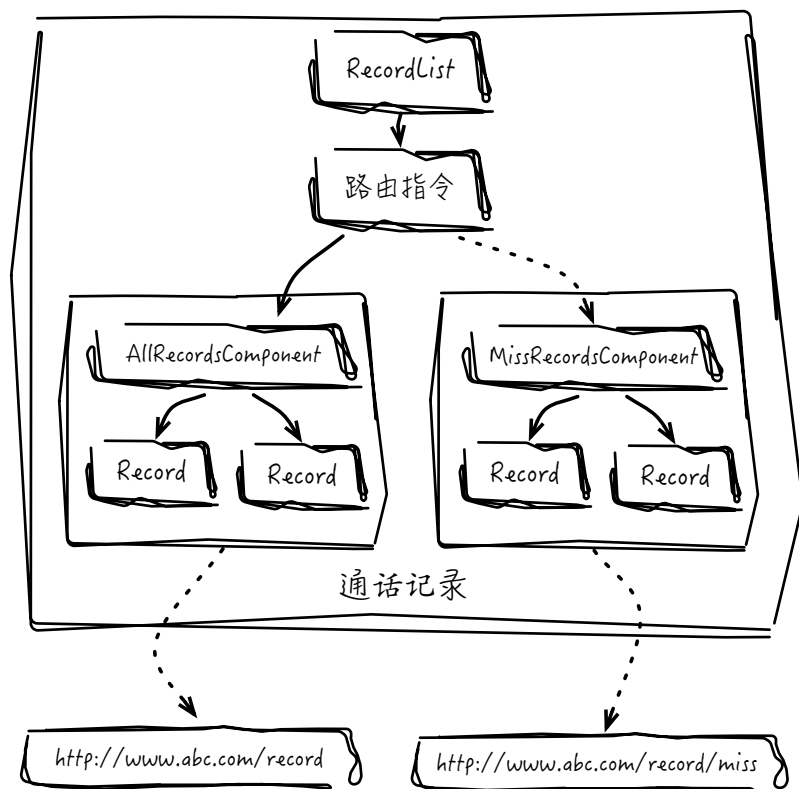


图 5-10 通话记录组件子路由功能

路由还支持路径参数，如 <http://www.abc.com/list/123>，其中 123 为联系人 ID，从而实现类似于 RESTful 风格的 URL 形式。另外，在同一层节点上还可以放置多个路由指令，实现从属路由功能。关于更多、更强大的路由功能将会在第 11 章中进一步剖析。

5.2 应用模块

在上面章节中，读者了解了 Angular 应用中的六个主要组成部分，那么这些不同的组成部分是如何组织起来，构成一个完整的功能单元甚至是完整的应用呢？Angular 引入了模块机制，对某些特定的功能特性进行封装，可能包含若干组件、指令、服务等，甚至拥有独立的路由配置，其关系如图 5-11 所示。

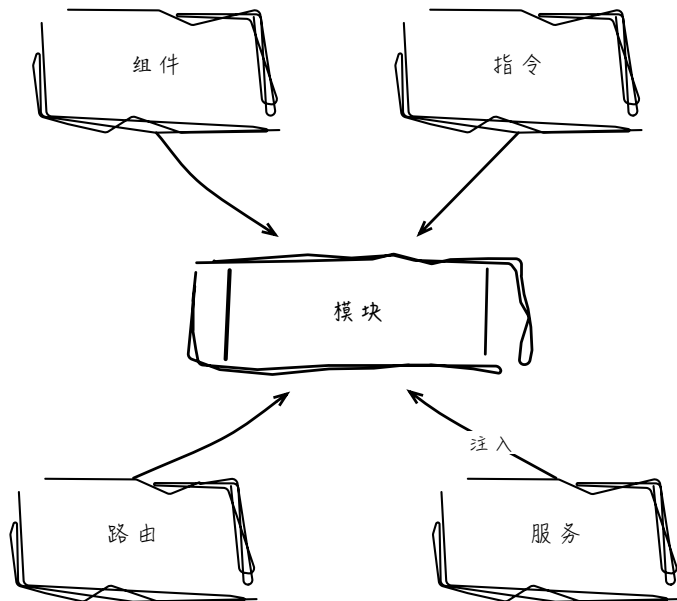


图 5-11 模块内关系图

每个 Angular 应用都至少有一个模块，一般需要有一个模块作为应用的入口，这个入口模块称为根模块（Root Module），通过引导运行根模块来启动 Angular 应用（这个引导过程下文会展开讲述）。虽然开发者可以把整个应用的逻辑（指所有的组件指令及服务）都封装到这个根模块里，但这通常并不是好的设计。比较推荐的做法是把应用功能分区设计，不同的功能特性由不同的独立模块负责，这种设计显然耦合性更低，更容易维护。在 Angular 中，除根模块外，其他的模块类型有：封装某个完整功能的特性模块（Feature Module）、封装一些公共构件的共享模块（Shared Module），以及存放应用级别核心构件的核心模块（Core Module）。

从图 5-12 可以看到，通过将特性模块导入到根模块里即可实现对该特性功能的引入，而模块间如何交互无须开发者关心，Angular 已经处理好了。这种交互关系对于不同的模块构件各不相同。

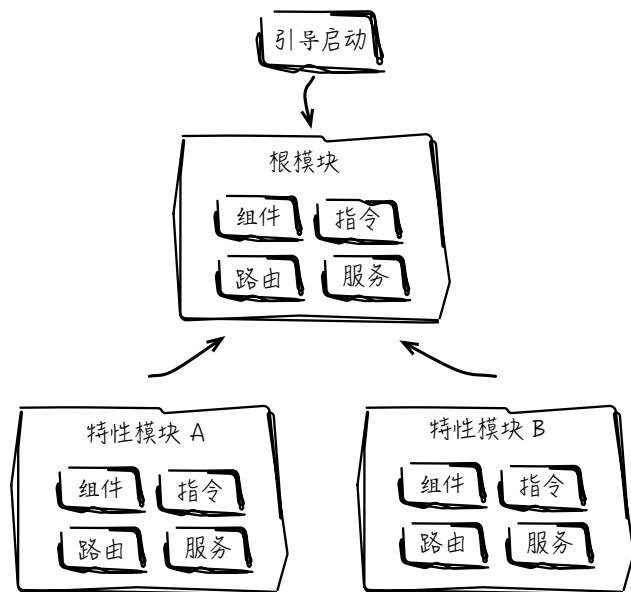


图 5-12 模块间关系图

- 路由：特性模块也可以自带路由配置，当特性模块导入到根模块后，特性模块的路由配置会自动与根模块里的路由配置合并。
- 组件和指令：在默认情况下，模块内的组件和指令是私有的。也就是说，特性模块 A 被导入到根模块后，根模块依然不能使用特性模块 A 里的组件和指令，除非特性模块 A 里显式暴露了某些组件或指令，这些暴露的组件或指令相当于模块的 API。
- 服务：服务的处理则有些特殊，通过依赖注入机制，服务同样可以注入到模块里，但跟组件里的依赖注入的作用域并不相同。注入到组件里的服务只能使用在该组件及其子组件上，而注入到模块里的服务在整个应用里均能使用，因为所有模块都共享着同一个应用级别的根注入器。这种机制似乎有点违背模块的封装性，到底这种设计是否合适，当有命名冲突时又是怎么解决的，这些疑问会在后续第 10 章中详细解答。

Angular 已经封装了不少常用的特性模块，如：

- `ApplicationModule`——封装一些与启动相关的工具。
- `CommonModule`——封装一些常用的内置指令和内置管道等。
- `BrowserModule`——封装在浏览器平台运行时的一些工具库，同时将 `Common-`

Module 和 AppModule 打包导出，所以通常在使用时引入 BrowserModule 就可以了。

- FormsModule 和 ReactiveFormsModule——封装与表单相关的组件指令等。
- RouterModule——封装与路由相关的组件指令等。
- HttpClientModule——封装与网络请求相关的服务等。

上述已提及，Angular 通过引导运行根模块来启动应用，引导方式有两种：动态引导和静态引导。要理解两者的区别，先来看看 Angular 应用的启动过程——Angular 应用在运行前，都需要经过编译器对模块、组件等进行编译，编译完成后才开始启动应用并渲染界面。

动态引导和静态引导的区别就在于编译的时机不同，动态引导是将所有代码加载到浏览器后，在浏览器中进行编译，即 JiT 编译；而静态引导是将编译过程前置到开发时的工程打包阶段，加载到浏览器的将是编译后的代码，称为 AoT 编译。

假设根模块为 AppModule，动态引导的示例代码如下：

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

动态引导从 platformBrowserDynamic 函数启动，该函数是从 @angular/platform-browser-dynamic 文件模块（关于 Angular 文件模块将在下一节中讲述）导入的。动态引导启动的模块 AppModule 即是我们编写的模块。再来看看静态引导的示例代码：

```
import { platformBrowser } from '@angular/platform-browser';
import { AppModuleNgFactory } from './app.module.ngfactory';
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

静态引导从 platformBrowser 函数启动，这个函数是从 @angular/platform-browser 文件模块导入的，跟动态引导的不是同一个。静态引导启动的是 AppModuleNgFactory 模块，这是 AppModule 经过编译处理后生成的模块（app.module 文件编译后生成 app.module.ngfactory 文件）。由于省去了浏览器编译这个步骤，因此应用启动的速度也会更快。

Angular CLI 包含的构建工具已经非常好地支持 AoT 编译，在开发中只需要按照 JiT 方式进行代码编写，构建时打开 AoT 编译选项，如 ng build --aot，即可完成如文件编译及静态引导等这些 AoT 处理过程。JiT 编译开发流程简单明了，但性能欠佳，仅适合在开发阶段使用；而 AoT 编译性能提升明显，推荐使用。



构建工具在不断迭代升级中，在未来的迭代版本中将会支持在开发阶段实施 AoT 编译。

5.3 源码结构介绍

Angular 是基于 TypeScript 编写的，TypeScript 又是 ES 6 的超集，而 ES 6 给开发者带来的一个新特性是文件级别的模块功能。利用这个特性，整个 Angular 项目的源码是基于 ES 6 模块来组织的。注意这里的模块和上一节提到的模块并不是同一个概念，上节提到的是应用级别的模块，是以功能特性为划分依据的；而本节的模块是语言级别的，是以物理文件或文件夹为划分依据的。GitHub 源码结构如图 5-13 所示。

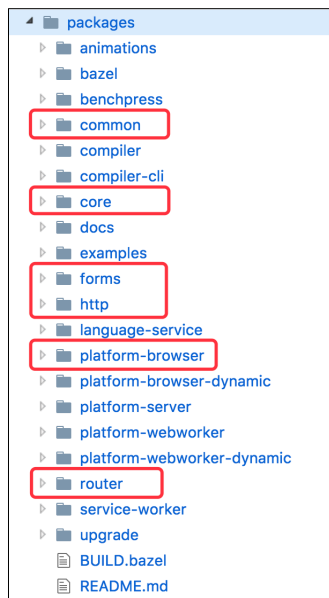


图 5-13 GitHub 源码结构图



在实际的 npm 包使用中，使用 @angular 作为命名空间来引入 Angular 模块，如 @angular/common。源码在发布至 npm 前会经过构建，把 packages 文件夹下的模块打包成 @angular 前缀。

在图 5-13 中，用方框标记的为常用的一级模块（一级文件夹），下面对主要的模块进行介绍。

- `packages/core`：存放核心代码，如变化监测机制、依赖注入机制、渲染等，核心功能的实现、装饰器（`@Component`、`@Directive` 等）也会存放到这个模块中。
- `packages/common`：存放一些常用的内置指令和内置管道等。
- `packages/forms`：存放与表单相关的内置组件及内置指令等。
- `packages/http`：存放与网络请求相关的服务等。在 Angular 4.3 以上版本中已有网络请求工具的替代方案，存放在 `packages/common/http` 里，第 9 章会进行详细讲解。
- `packages/router`：存放与路由相关的组件和指令等。
- `packages/platform-<x>`：存放的是与引导启动相关的工具。Angular 支持在多个平台下运行，不同的平台都有对应的启动工具，这些启动工具会被封装到不同的模块里，如服务端渲染这个场景的启动工具存放在 `packages/platform-server` 下，浏览器的启动工具则存放在 `packages/platform-browser` 下。

这些语言级别的模块和应用级别的模块非常相似，实际上它们是有关联的，如 `CommonModule` 模块本身就存放在 `packages/common` 里，当开发者需要引用 `packages/common` 里的诸多指令或者组件时，只需引入 `CommonModule` 即可。`CommonModule` 的作用是打包 `packages/common` 下零散的组件指令并作为该模块的 API 暴露出来，方便开发者一次性引入。

关于其他模块，有兴趣的读者可以到 Angular 的 GitHub 库中查阅。

5.4 小结

Angular 的各个组成部分以组件作为桥梁关联起来，对组件的深度剖析将作为本书深入讲解的开篇；之后是与组件密切相关的模板章节，它囊括了 Angular 大部分内置的模板语法，仔细阅读完后可对 Angular 的视图特性有一个系统的认识；接着，指令、服务、依赖注入及路由这四个相对比较独立的概念会分别开辟新章节讲述；最后会加入测试这个特殊章节，测试是保证应用质量的关键环节，该章节会详细介绍 Angular 的各个部分是如何实施自动化测试的。

这就是 Angular 的总体架构，实际上它已不仅仅是简单的框架，更像是一个平台。精心的架构设计、成熟的 Angular 生态、对标准的拥抱，还有 Google 和微软的联手支持，这些都给了开发者足够的信心，Angular 将会是一个非常棒的平台！

6 组件

组件（Component）是构成 Angular 应用的基础和核心，了解它如何工作是非常重要的。通俗地说，组件用来包装特定的功能，应用程序的有序运行依赖组件之间的协同工作。举个现实中的例子，一辆汽车包含各种各样的部件，比如发动机、变速箱、轮胎等，这些可以理解为组成汽车的组件，当然像发动机这种部件还不是最小粒度的组件，它本身也是由众多小组件组合而成的。

通过本章的内容，读者可以了解到组件化的发展和 Web Component 标准是如何形成的，以及 Angular 如何向 Web Component 靠齐；接着从如何创建组件开始，到组件的构成，以及组件和模块的关系，从基础到深入来学习组件的元数据、生命周期、组件交互及变化监测机制等内容。

6.1 概述

下面来了解组件的概念是如何产生的，以及组件化又是如何发展的。

6.1.1 模块化介绍

谈到组件化，先要了解模块化。在 Node.js 中，模块就是一个文件，引入一个文件就是简单地通过 `require('filePath')` 引入的，其中 `filePath` 是文件名称或路径。在前端领域，也衍生出不少模块 XX 化的规范，比如 AMD。不过前端领域发展得很快，因为还未形成

一种统一的规范,在写作本书时,AMD都已经不那么“流行”了。后来比较流行的如ES 6引入模块的方式,与Node.js比较相似,通过`import { SomeClassName } from 'filePath'`来引入一个模块。



AMD 全称是 Asynchronous Module Definition, 它采用非同步加载方式, 允许指定回调函数操作。著名的 RequireJS 是 AMD 规范的实现之一。

在早期的模块化工具中,多数只是针对JavaScript文件部分做了处理,比如RequireJS、Browserify,而缺少对CSS、HTML等文件进行管理的工具。后来逐渐形成按模块划分的概念,对比传统的按资源目录划分,从逻辑的意义上来说似乎更加合理,让模块更独立,方便维护。

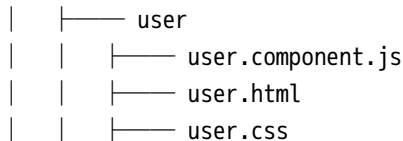
- 按资源划分

```
|— project
|   |— css
|   |— js
|   |— img
|   |— template
|   |— index.html
```

- 按模块划分

```
|— project
|   |— shop
|       |— shop.component.js
|       |— shop.html
|       |— shop.css
|   |— user
|       |— user.component.js
|       |— user.html
|       |— user.css
|   |— index.html
```

通过将.js、.css、.html文件按逻辑模块划分后,使得逻辑结构更加清晰,这样逐步便形成了组件的概念。读者可以简单理解为,前端中的组件就是一堆为了实现同一业务逻辑的代码文件的组合。比如在上面按模块划分的结构中,用户模块user便是一个实现用户相关逻辑的组件,它将相关的文件都存放到同一个目录中,独立性非常强,如果要删除这个模块,直接删除文件目录即可。其目录结构如下:



前端领域发展得很快,各种库、框架层出不穷,例如 AngularJS 1.x、Backbone、Ember、React、Vue 等;工程化的工具也越来越多,例如 Browserify、Grunt、Gulp、Webpack 等。近几年在技术领域,恐怕也只有在前端领域才出现了这么多创新变化的景象。这一方面说明前端领域的创造力很强;另一方面也正说明了前端领域缺乏标准,每种库、框架都有自己的一套组件化方式。

6.1.2 组件化标准

W3C 为了统一组件化的标准方式,提出了 Web Component 标准。通过标准化的非侵入方式封装组件,每个组件都包含自己的 HTML、CSS、JavaScript 代码,并且不会对页面上的其他组件产生影响。



关于 Web Component 标准的最新状态,可以在 W3C 官网进行了解: https://www.w3.org/standards/techs/components#w3c_all。

Web Component 是由一些新技术构成的,还提供了浏览器原生的 UI 组件标准。因为是原生的,所以不需要引入任何外部依赖。要使用一个已有的 Web Component,只需要添加一句导入的声明即可,类似于:

```
<link rel="import" href="example.html" />
```

Web Component 标准包括如下四个重要的概念。

- 自定义元素: 这个特性允许开发者创建自定义的 HTML 标签和元素,每个元素都有属于自己的脚本和样式。
- 模板: 模板允许开发者使用 `<ng-template>` 标签预先定义一些内容,但并不随页面加载而渲染,而是可以在运行时使用 JavaScript 来初始化它。
- Shadow DOM: 通过 Shadow DOM 可以在文档流中创建一些完全独立于其他元素的 DOM 子树,这个特性可以让开发者开发一个独立的组件,并且不会干扰其他 DOM 元素。
- HTML 导入: 一种在 HTML 文档中引入其他 HTML 文档的方法,用于导入 Web Component 的机制。

这里通过“定义一个 Hello 组件”的简单示例，说明 Web Component 标准的四个概念。首先，如图 6-1 所示，创建在页面上显示“Hello Web Component!”字样的应用，并把字设置为红色。



图 6-1 Web Component Hello 效果图

需要注意的是，在写作本书时，主流浏览器仍未完全实现 Web Component 标准，其中 Chrome 浏览器对该标准支持度最高。根据 Web Component 标准实现的 Hello 组件，示例代码如下：

```
<!-- hello.html -->
```

```
<!-- 模板：此处使用到模板的特性，并自定义了一些内容，这些内容并不会随着页面的  
      加载而加载 -->
```

```
<ng-template id="hello-template">  
  <style>  
    h1 { color: red }  
  </style>  
  <h1>Hello Web Component!</h1>  
</ng-template>
```

```
<script>
```

```
  // 指向导入文档，即下面的 index.html  
  var indexDoc = document;
```

```
  // 指向被导入文档，即当前文档 hello.html  
  var helloDoc = (indexDoc._currentScript || indexDoc.currentScript).ownerDocument;
```

```
  // 获得上面的模板  
  var tmpl = helloDoc.querySelector('#hello-template');
```

```
  // 创建一个新元素的原型，继承自 HTMLElement  
  var HelloProto = Object.create(HTMLElement.prototype);
```

```
// 设置 Shadow DOM 并将模板的内容复制进去
HelloProto.createdCallback = function() {
  var root = this.createShadowRoot();
  root.appendChild(indexDoc.importNode(tmpl.content, true));
};

// 注册新元素
/**
 * 自定义元素：通过浏览器原生的方法，注册一个叫作 hello-component 的标签
 * 注：在撰写本书之时，document.registerElement() 方法已经废弃了，取而代之的是
 *     customElements.define() 方法，但目前并未有浏览器支持，本例仅供参考
 */
var Hello = indexDoc.registerElement('hello-component', {
  prototype: HelloProto
});
</script>
```

假设 Hello 组件的代码片段保存在某处的 `hello.html` 中，那么在页面中可以通过 HTML 导入的方式引入，最终可以在页面上通过自定义的 HTML 标签 `<hello-component>` 使用它。示例代码如下：

```
<!-- index.html -->

<!doctype html>
<html>
<head>
  <meta charset="utf-8">

  <!-- HTML 导入：通过此特性将外部组件以 HTML 文档的方式导入到主文档流中 -->
  <link rel="import" href="hello.html">
</head>
<body>
  <hello-component></hello-component>
</body>
</html>
```

`hello.html` 就像一个普通的 HTML 文档，这意味着它可以像普通的 HTML 文档那样引入其他脚本或样式表。比如希望在 Hello 组件中引入 Bootstrap 的样式库，那么就可以在 `hello.html` 开头加上 `<link rel="stylesheet" href="bootstrap.css">`。这个特性使得

开发者能够将多个文档（组件）封装成一个文档（组件）提供他人使用。关键的一点是，通过 HTML 导入的特性，使得异步加载模块及模块执行变得更加容易。

最终 `<hello-component></hello-component>` 在页面上以 Shadow DOM 的方式渲染出来，如图 6-2 所示。

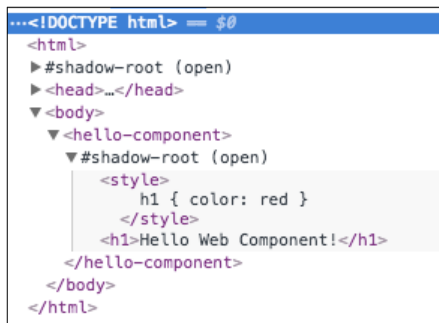


图 6-2 Hello 组件的 Shadow Root

浏览器原生 Shadow DOM 的特性满足了 Web Component 组件之间互不干扰的要求，通过创建一棵 `#shadow-root` 子树，将该组件节点与文档流中的其他 DOM 节点隔离开，彻底解决了样式冲突、全局变量污染等问题。

Google 官方出品的 Polymer 框架则比较接近 Web Component 的写法，它是面向未来框架的。同样是出自 Google 的 Angular 与 Web Component 也有几分相似，虽然 AngularJS 1.x 版本也支持模板和自定义标签，但 Angular 的组件化程度比 AngularJS 1.x 更加彻底，而且与 Web Component 标准更接近。



在写作本书时，只有 Chrome 和 Opera 浏览器对 Web Component 的支持度比较高，哪怕是很简单的使用 Web Component 标准写的 hello-world 代码，其他浏览器也不能运行。

6.1.3 Angular 的组件

起初 Angular 是以 Web Component 标准为蓝本进行设计的，在 Angular 中引入了视图包装（ViewEncapsulation）的概念，允许通过设置 `ViewEncapsulation.Native` 选项来使用原生的 Shadow DOM。详细内容会在本章的“扩展阅读”部分讲述。

除此之外，Angular 还支持模板、自定义标签、异步加载组件等。Angular 的组件是自描述的——可以和宿主元素交互，知道如何及何时渲染自己，可配置注入服务，有明

确的 Input & Output 定义（注入与服务是后续章节中的内容，宿主元素、Input 和 Output 在本章后续内容中讲述）。所有的 Angular 组件都可以独立存在，这意味着任何 Angular 组件都可以作为根组件被引导，也可以被路由加载，或者在其他组件中使用。不过，一个组件不能单独被启动，它必须被包装到模块（NgModule）里，然后通过 Bootstrap 模块接口引导入口模块来启动 Angular 应用。

在第 5 章中，我们学习了模块的概念。有了模块，开发者可以更好地将应用按功能特性组织成一个个模块集合，从而降低应用的耦合性。

组件是 Angular 应用的最小逻辑单元，模块则是在组件之上的一层抽象。组件及其他部件如指令、管道、服务、路由等都可以被包含到一个模块中，外部通过引用这个模块来使用一系列封装好的功能。读者可以将 Angular 应用想像成一棵树，组件是这棵树的叶子，模块便是这棵树的树枝，每个 Angular 应用都必须要有有一个根模块（树干），并且在根模块中必须通过 Bootstrap 指定根组件，用于启动 Angular 应用。

本章接下来的内容会结合代码演示，以及第 4 章中的通讯录例子，对组件及其相关的概念进行详细讲解。

6.2 组件基础

6.2.1 创建组件的步骤

学习 Angular 的组件，首先要知道怎么创建它。创建组件很简单，Angular 提供了很方便的方法。创建 Angular 组件可以通过以下三个步骤。

- （1）从 @angular/core 中引入 Component 装饰器。
- （2）建立一个普通的类，并用 @Component 修饰它。
- （3）在 @Component 中设置 selector 自定义标签和 template 模板。

此处通过创建一个显示名称和电话的联系人卡片来说明组件的创建方法。联系人卡片 ListItemComponent 组件的示例代码如下：

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-list-item',
  template: `
    <div>
      <p>张三</p>
```

```
        <p>13800000000</p>
      </div>
    、
  })
  export class ListItemComponent {}
```

以上代码创建了一个最简单的 Angular 组件。使用这个组件，需要在 HTML 中添加 `<app-list-item>` 自定义标签，然后 Angular 便会在此标签中插入在 `ListItemComponent` 组件中指定的模板。



当然，这么做还不能让 Angular 渲染这个组件，需要结合模块来启动应用才能运行起来。相关概念会在本章后续内容中讲解。

在 HTML 中使用 `ListItemComponent` 组件的示例代码如下：

```
<div>
  <app-list-item></app-list-item>
</div>
```

最终上述 HTML 的结构会被渲染成：

```
<div>
  <app-list-item>
    <div>
      <p>张三</p>
      <p>13800000000</p>
    </div>
  </app-list-item>
</div>
```

6.2.2 组件的基础构成

组件的基础构成如图 6-3 所示。

在组件的基础构成中，有以下几个关键知识点。

- 组件装饰器（Component Decorator）：每个组件类都必须用 `@Component` 进行修饰才能成为 Angular 组件。
- 组件元数据（Component Metadata）：selector、template。
- 模板：每个组件都会关联一个模板，这个模板最终会渲染到页面上，页面的这个 DOM 元素就是此组件实例的宿主元素。

- 组件类：组件实际上也是一个普通的类，组件的逻辑都在组件类里定义并实现。

```
import { Component } from '@angular/core';

@Component({ ..... 1
  selector: 'app-list-item',
  template: ` ..... 2
    <div>
      <p>张三</p>
      <p>13800000000</p>
    </div> ..... 3
  `
}) ..... 4
export class ListItemComponent {}
```

图 6-3 组件的基础构成



组件的元数据并不只有 selector、template，更多的元数据在本章后续内容中会继续讲解。

组件装饰器

@Component 是 TypeScript 的语法，它是一个装饰器，任何一个 Angular 的组件类都会用这个装饰器修饰，如果移除了这个装饰器，它将不再是 Angular 的组件。

由于浏览器不能直接解释 TypeScript 代码，最终组件的代码会通过 TypeScript 解析器转换成 JavaScript 代码。转换后的代码如下：

// item.component.ts 转换成 JavaScript 代码示例

```
var ListItemComponent = (function () {
  function ListItemComponent() {}
  ListItemComponent = __decorate([
    core_1.Component({
      selector: 'app-list-item',
      template: `
        <div>
          <p>张三</p>
          <p>13800000000</p>
        </div>
      `
    })
  ], ListItemComponent);
})();
```



```

    })
    __metadata('design:paramtypes', [])
  ], ListItemComponent);
  return ListItemComponent;
}());

```

可以看到，Angular 的 `@Component` 会被转换成一个 `__decorate()` 方法，元数据的定义通过 `core_1.Component` 传入，将 `ListItemComponent` 这个类“装饰”起来，使得 `ListItemComponent` 具有装饰器里定义的元数据属性。所以装饰器可以理解为对组件封装的语法糖，方便开发者编写 Angular 的组件。

组件元数据

在 `ListItemComponent` 组件中的 `@Component` 装饰器部分，使用到了大部分组件都需要的元数据——用于定义组件标签名的 `selector`、用于定义组件宿主元素模板的 `template`（`template` 是用于定义内联模板的，Angular 还提供了 `templateUrl`，可以引用外联模板）。除此之外，组件一般都会有自己定义的样式，Angular 提供了 `styles` 和 `styleUrls` 来定义内联样式或引用外联样式。内联模板的示例代码如下：

```

// item.component.ts
@Component({
  selector: 'app-list-item',
  template: `
    <div>
      <p>张三</p>
      <p>13800000000</p>
    </div>
  `,
})

```

selector

`selector` 用于定义组件在 HTML 代码中匹配的标签，它将成为组件的命名标记。在通常情况下都需要设置 `selector`，在特殊情况下也可以忽略，不指定时设置默认为匹配 `div` 元素，但建议不要这样做，因为这样组件会无法准确定位 DOM 中的元素。`selector` 的命名方式建议使用“烤肉串式”，即采用小写字母并以“-”分隔，例如“contact-app”“hello-world”等。

在 `ListItemComponent` 组件中，它的 `selector` 设置为 `app-list-item`，那么 Angular 便会通过查找 DOM 上的 `app-list-item` 元素，将组件的内容显示在页面上。那么组件的内容又是怎么指定的？这就要用到 `template` 或 `templateUrl` 了。

template

`template` 用于为组件指定一个内联模板。关于模板中的语法内容，请参见后面的“模板”章节。如果要写内联模板，建议使用 ES 6 的多行字符串“````”（两个反引号）语法，这样能够创建多行模板。示例代码如下：

```
@Component({
  template: `
    <div>
      <div>Inner Div</div>
    </div>
  `,
})
```

templateUrl

`templateUrl` 用于为组件指定一个外部模板的 URL 地址。示例代码如下：

```
@Component({
  templateUrl: 'app/list/item.component.html'
})
```

每个组件只能指定一个模板，可以使用 `templateUrl` 或者 `template` 的引入方式。本书更推荐使用 `templateUrl`，因为无论如何，组件的内容都会由不少的 DOM 元素组成，在 TypeScript 文件里写一长串 HTML 代码不是一个好主意，除非模板只有一两行极简的代码，并且不会发生变化。



在本书的某些代码示例中，使用 `template` 仅仅为了方便演示。

在本书 4.2 节“通讯录例子”中，每个组件都使用了外联模板，比如在 `item.component.ts` 中使用了 `templateUrl`，为 `ListItemComponent` 组件指定了一个 `item.component.html` 模板，它的内容如下：

```
<!-- item.component.html -->

<a (click)="goDetail(contact.id)">
```

```

<div class="contact-info">
  <label class="contact-name">{{ contact.name }}</label>
  <span class="contact-tel">{{ contact.telNum }}</span>
  <i class="contact-to-detail"></i>
</div>
</a>
```

然后，Angular 会在 DOM 树中搜索 `<app-list-item></app-list-item>` 这个位置，最后将 `item.component.html` 的内容添加到 DOM 节点上。

styles

`styles` 用于为组件指定内联样式。示例代码如下：

```
@Component({
  styles: [
    \
    li:last-child{
      border-bottom: none;
    }
    \
  ]
})
```

styleUrls

`styleUrls` 用于为组件指定一系列外联样式表文件。示例代码如下：

```
@Component({
  styleUrls: ['app/list/item.component.css']
})
```



`styles` 和 `styleUrls` 允许同时指定。如果同时指定，`styles` 中的样式会先被解析，然后才会解析 `styleUrls` 中的样式。换句话说，`styles` 的样式会被 `styleUrls` 的样式覆盖。另外，也可以在模板的 DOM 节点上直接写样式，也就是模板内联样式（Template Inline Style），它的优先级是最高的。我们强烈建议只使用 `styles` 或 `styleUrls` 的其中一个来设置样式。另外，也建议像选择 `templateUrl` 那样，使用 `styleUrls` 引入外部样式表文件，这样代码结构更清晰、更易于管理。

此外，通过 `styles` 和 `styleUrls` 指定组件的样式，Angular 会在模板 DOM 中添加自定义的节点属性，以此来形成属于这些样式在组件中独有的作用域，避免了 CSS 样式命名的污染问题。

细心的读者可以注意到，`styles` 和 `styleUrls` 都是复数，所以要使用数组来承载样式表字符串或样式表文件路径字符串。Angular 允许使用多个样式表文件来同时渲染一个组件。

开发者也可以使用一些 CSS 预处理器来更好地编写 CSS 代码，如 SCSS 等。在使用 Angular CLI 创建项目时，加入 `--style` 参数即可。执行命令如下：

```
ng new my-project --style=scss
```

如果需要修改已创建好的项目的样式处理器，可使用 `ng set` 命令，如下所示：

```
ng set defaults.styleExt scss
```

创建或设置好项目后，在组件里直接引用 `.scss` 样式文件即可。示例代码如下：

```
@Component({
  styleUrls: ['app/list/item.component.scss'] // .scss 文件
})
```

模板

在上面的章节中，介绍过 `template` 和 `templateUrl` 元数据。每个组件都必须设置一个模板，Angular 才能将组件内容渲染到 DOM 上，这个 DOM 元素就叫作**宿主元素**。上文也提到组件是自描述的，其中一点便是组件可以与宿主元素交互，交互的形式包括：

- 显示数据。
- 双向数据绑定。
- 监听宿主元素事件，以及调用组件方法。

显示数据

在模板中，可以使用插值语法“`{{}}`”来显示组件的数据，这和 AngularJS 1.x 版本是一致的。例如在 `ListItemComponent` 组件中，可以把“张三”和“电话”都通过组件类的成员变量来替代。示例代码如下：

```
// item.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-list-item',
  template: `
    <div>
      <p>{{name}}</p>
      <p>{{phone}}</p>
    </div>
  `,
})
export class ListItemComponent {
  name: string = '张三';
  phone: string = '13800000000';
}
```

双向数据绑定

Angular 提供了双向数据绑定的功能，这在控制用户输入时很有用处，双向绑定使用的是形如 [(ngModel)]="property" 的语法。为了方便说明，我们对 ListItemComponent 组件进行改造，示例代码如下：

```
// item.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-list-item',
  template: `
    <div>
      <input type="text" value="{{name}}" [(ngModel)]="name"/>
      <p>{{name}}</p>
      <p>{{phone}}</p>
    </div>
  `,
})
export class ListItemComponent {
  name: string = '张三';
  phone: string = '13800000000';
}
```

在上面的例子中，输入框绑定了 `name` 属性，默认显示“张三”。当输入框的内容发生改变时，Angular 的双向绑定机制便会将输入框的内容同步更新到 `name` 属性，并且会同步显示到 `<p>{{name}}</p>` 上。

监听宿主元素事件，以及调用组件方法

Angular 提供了非常方便的事件绑定的方式，读者可以在通讯录例子的 `list.component.html` 文件中看到 `(click)` 这样的用法。`()` 是 Angular 提供的事件绑定语法糖，通过 `(eventName)` 的方式可以轻易地响应 UI 事件，如 `list.component.html` 中事件绑定的代码：

```
<h3>所有联系人<i (click)="addContact()"></i></h3>
```

调用组件方法一般和监听事件一起进行，像上面提到的 `(click)="addContact()"`，就是在用户单击了该 DOM 元素时调用组件的 `addContact()` 方法的。



关于模板的更多语法细节，将会在后面的第 7 章中进行详细讲解。

6.2.3 组件与模块

通常组件不会独立存在，而是通过与其他组件协作来完成一个完整的功能特性。在 Angular 中，这样的功能特性通常会封装到一个模块里。

模块是在组件之上的一层抽象，组件及指令、管道、服务、路由等都能通过模块来组织。下面来看看模块及组件是如何协作的。

模块的构成

Angular 提供了 `@NgModule` 装饰器来创建模块。一个应用可以有多个模块，但有且只有一个根模块（Root Module），其他模块叫作特性模块（Feature Module）。根模块是启动应用的入口模块，根模块必须通过 `bootstrap` 元数据来指定应用的根组件，然后通过 `bootstrapModule()` 方法来启动应用。

回想一下上文创建的 `ListItemComponent` 组件，示例代码如下：

```
// item.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-list-item',
```

```
template: `
  <div>
    <p>张三</p>
    <p>13800000000</p>
  </div>
`,
})
export class ListItemComponent {}
```

为了启动这个组件，需要建立一个根模块（假设命名为 `AppModule`）并将它保存为 `app.module.ts`，然后通过 `@NgModule` 的 `bootstrap` 元数据指定 `ListItemComponent` 组件。示例代码如下：

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ListItemComponent } from './list/item.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [ListItemComponent],
  bootstrap: [ListItemComponent]
})
export class AppModule {}
```

最后创建一个 `app.ts`，利用 `platformBrowserDynamic().bootstrapModule()` 方法来启动这个根模块，这样 Angular 应用就能运行起来，并将 `ListItemComponent` 组件的内容展示到页面上。示例代码如下：

```
// app.ts
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

刚才提到了 `bootstrap` 这个元数据，它用于指定应用的根组件。`NgModule` 的主要元数据如下。

- **declarations**：用于指定属于这个模块的视图类（View Class），即指定哪些部件组成了这个模块。Angular 有组件、指令和管道三种视图类，这些视图类只能属于一个模块。注意，不要再次声明属于其他模块的类。

- **exports**: 导出视图类。当该模块被引入外部模块中时, 这个属性指定了外部模块可以使用该模块的哪些视图类, 所以它的值类型跟 **declarations** 一致 (组件、指令和管道)。
- **imports**: 引入该模块依赖的其他模块或路由, 引入后模块里的组件模板才能引用外部对应的组件、指令和管道。
- **providers**: 指定模块依赖的服务, 引入后该模块中的所有组件都可以使用这些服务。

视图类引入

NgModule 提供了 **declarations** 元数据来将指令、组件或管道等视图类引入到当前模块中。在上面例子的 AppModule 中, 通过 **declarations: [ListItemComponent]** 将 ListItemComponent 组件引入到 AppModule 模块中, 使得所有属于 AppModule 模块的其他组件的模板都可以使用 `<app-list-item>`。不过, 在这个例子中只有一个组件, 接下来看看通讯录例子中的 `app.module.ts` 文件。示例代码如下:

```
// app.module.ts
@NgModule({
  declarations: [
    AppComponent,
    ListComponent,
    ListItemComponent,
    DetailComponent,
    CollectionComponent,
    EditComponent,
    HeaderComponent,
    FooterComponent,
    PhonePipe,
    BtnClickDirective
  ]
  // ...
})
export class AppModule {}
```

其中 PhonePipe 是管道, BtnClickDirective 是指令, 其他的均是组件。例如, 在 ListComponent 组件的模板代码 `list.component.html` 中, 使用到了 HeaderComponent、FooterComponent 及 ListItemComponent 这三个组件, 这个时候必须在 ListComponent 所属的模块 (即 AppModule) 中, 通过 **declarations** 引入 HeaderComponent、FooterComponent 及 ListItemComponent 后才能在模板中使用它们。

ListComponent 组件的模板的示例代码如下：

```
<!-- list.component.html -->
<!-- 在组件中指定了 HeaderComponent, 才能使用 my-header 标签 -->
<my-header title="所有联系人" [isShowCreateButton]="true"></my-header>
<ul class="list">
  <li *ngFor="let contact of contacts">
    <!-- 在组件中指定了 ListItemComponent, 才能使用 list-item 标签 -->
    <app-list-item [contact]="contact" (routerNavigate)="routerNavigate($event)"></app-list-item>
  </li>
</ul>
<!-- 在组件中指定了 FooterComponent, 才能使用 my-footer 标签 -->
<my-footer></my-footer>
```

细心的读者可能会发现，ngFor 是 Angular 的内置指令，似乎没有引入就可以使用，但实际上我们在引入 BrowserModule 的时候就已经引入了常用的内置指令，其中就包括 ngFor。

导出视图类，以及导入依赖模块

有时候模块中的组件、指令或管道，可能也会在其他模块中使用。可以使用 exports 元数据对外暴露这些组件、指令或管道，这里还是通过通讯录例子来说明 exports 的用法。

想象一下手机通讯录及发短信的场景，再回想一下刚才的联系人信息 ListItemComponent 组件，联系人信息可以从联系人列表中进入查看，也可以从短信的界面进入查看。假设通讯录模块叫作 ContactModule，短信模块叫作 MessageModule，然后联系人信息组件被封装在 ContactModule 模块中，此时联系人信息组件便需要被 ContactModule 模块导出（exports），才能够被其他模块使用。

在 contact.module.ts 文件代码中，通过 exports 元数据，将希望在其他模块中使用的 ListItemComponent 组件导出。示例代码如下：

```
// contact.module.ts
import { NgModule } from '@angular/core';
import { ListItemComponent } from './list/item.component';

@NgModule({
  declarations: [ListItemComponent],
  exports: [ListItemComponent] // 导出组件
```

```
)  
export class ContactModule {}
```



contact.module.ts 和 message.module.ts 这两个文件在 GitHub 的示例中并不存在，此处仅用来方便描述。

在短信模块中，只需要将依赖的 ContactModule 模块引入，便可以在 MessageModule 模块的其他组件的模板中，使用 ContactModule 导出的 ListItemComponent 组件。示例代码如下：

```
// message.module.ts  
import { NgModule } from '@angular/core';  
import { ContactModule } from './contact.module';  
import { SomeOtherComponent } from './someother.component';  
  
@NgModule({  
  // 在 SomeOtherComponent 组件的模板中，可以使用 <app-list-item> 组件了  
  declarations: [SomeOtherComponent],  
  // 导入模块  
  imports: [ContactModule]  
})  
export class MessageModule {}
```

服务引入

服务通常用于处理业务逻辑及相关的数据。引入服务有两种方式：一种是通过 @NgModule 的 providers；另一种是通过 @Component 的 providers。

在通讯录例子的根模块 app.module.ts 文件代码中，通过 providers 元数据注入了自定义的 ContactService 服务。ContactService 是维护联系人数据的主服务，负责对联系人信息的相关操作。示例代码如下：

```
// app.module.ts  
import { ContactService } from './shared';  
// ...  
@NgModule({  
  // ...  
  providers: [ContactService],  
  bootstrap: [AppComponent]
```

```
})  
export class AppModule {  
  // ...  
}
```

这里通过 `@NgModule` 的 `providers` 来注入服务，所有被包含在 `AppModule` 中的组件都可以使用这些服务。同样，在组件中也可以用 `providers` 来引入服务，该组件及其子组件都可以共用这些引入的服务。



关于服务的更多内容请参阅第 9 章。

6.3 组件交互

在 6.2 节中我们学习了组件相关知识，接下来将学习组件交互。Angular 应用由各种各样的组件组成，这些组件形成了一棵组件树，数据可以在组件树里完成交互，组件间的交互包括父子组件的交互和一些非父子关系组件的交互。组件交互就是组件通过一定的方式来访问其他组件的属性或方法，从而实现数据双向流动。

组件交互有很多种方式可供选择，非父子关系的组件可通过服务来实现数据交互通信。



通过服务来实现组件间数据交互的方式将在第 9 章中详细讲述。

6.3.1 组件的输入、输出属性

Angular 提供了输入（`@Input`）和输出（`@Output`）语法来处理组件数据的流入流出，参照通讯录例子中 `item.component.ts` 及 `list.component.html` 的代码：

```
// item.component.ts  
export class ListItemComponent implements OnInit {  
  @Input() contact:any = {};  
  @Output() routerNavigate = new EventEmitter<number>();  
  // ...  
}  
  
<!-- list.component.html -->
```

```
<li *ngFor="let contact of contacts">
  <list-item [contact]="contact" (routerNavigate)="routerNavigate($event)">
  </list-item>
</li>
```

上面 `ListItemComponent` 组件的作用是显示单个联系人的信息。由于联系人列表数据是在 `ListComponent` 组件中维护的，在显示单个联系人时，需要给 `ListItemComponent` 传入单个联系人数据。另外，当单击单个联系人时，跳转到此联系人的明细信息处，需要子组件通知父组件进行跳转。因此，上述代码分别自定义了 `[contact]` 和 `(routerNavigate)` 的输入输出变量，用于满足上述功能。

被 `@Input` 修饰的 `contact` 变量属于输入属性，而被 `@Output` 修饰的 `routerNavigate` 则是输出属性，这里的“输入”“输出”是以当前组件的角度来说的。换句话说，`contact` 和 `routerNavigate` 分别是 `ListItemComponent` 组件的输入和输出属性。输出属性一般以事件的形式，将数据通过 `EventEmitter` 抛出去。

除使用 `@Input` 和 `@Output` 修饰外，还可以在组件的元数据中使用 `inputs`、`outputs` 来设置输入和输出属性，所设置的值必须为字符串数组，元素的名称需要和成员变量相对应。以下代码和上述代码是等价的：

```
// ...
@Component({
  // ...
  inputs: ['contact'], // 'contact' 匹配成员变量 contact
  outputs: ['routerNavigate'] // 'routerNavigate' 匹配成员变量 routerNavigate
})
export class ListItemComponent implements OnInit {
  contact:any = {};
  routerNavigate = new EventEmitter<number>();
  // ...
}
```

在理解了输入和输出属性的相关知识之后，下面将详细介绍组件间的数据交互。

6.3.2 父组件向子组件传递数据

父组件的数据通过子组件的输入属性流入子组件，在子组件中完成接收或拦截，以此实现了数据由上而下的传递。在 6.3.1 节中已经具体讲解了输入和输出属性（`@Input` 和 `@Output`）的具体使用方法，下面将使用通讯录例子中联系人及联系人详情页的代码示例来说明父组件是如何向子组件传递数据的。

父到子组件的数据传递

父组件 `ListComponent` 将获取到的联系人数据，通过属性绑定的方式流向子组件 `ListItemComponent`。我们先看父组件 `ListComponent` 的代码示例：

```
// list.component.ts
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'list',
  template: `
    <ul class="list">
      <li *ngFor="let contact of contacts">
        <app-list-item [contact]="contact"></app-list-item>
      </li>
    </ul>
  `
})
export class ListComponent implements OnInit {
  // ...
  this.contacts = data; // data为获取到的联系人数据
}
```

这里回顾一下之前的通讯录例子，在 `app.module.ts` 中已经通过 `@NgModule` 的元数据 `declarations` 将子组件 `ListItemComponent` 的实例引入到 `AppModule` 中，使得所有属于 `AppModule` 的其他组件都可以使用 `ListItemComponent` 组件，因此在父组件 `ListComponent` 中可直接引用该子组件。将每个联系人对象通过属性绑定的方式绑定到属性 `contact` 中来供子组件引用，数据由上而下流入子组件，在子组件中通过 `@Input` 装饰器进行数据的接收。子组件的示例代码如下：

```
// item.component.ts
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-list-item',
  template: `
    <div class="contact-info">
      <label class="contact-name">{{ contact.name }}</label>
      <span class="contact-tel">{{ contact.telNum }}</span>
    </div>
  `
})
```

```
    \n  })\n  export class ListItemComponent implements OnInit {\n    @Input() contact:any = {};\n    // ...\n  }
```

`ListItemComponent` 组件主要展示联系人姓名 (`name`) 和联系人电话号码 (`telNum`), 这两个属性包含在 `contact` 对象下, 其数据是通过装饰器 `@Input` 来获取来自父组件的 `contact` 对象的, 数据由父组件流出, 在子组件中通过输入属性 (`@Input`) 完成数据的接收。这样就简单地实现了父子组件的数据交互, 即通过使用属性绑定实现了从父组件向子组件传递数据的目的。

前面也讲到 `Angular` 应用是由各种各样的组件组成的, `Angular` 会从根组件开始启动, 并解析整棵组件树, 数据以由上而下的方式流向下一级子组件。不过需要注意的是, 目标属性必须通过输入属性 (`@Input`) 明确的标记修饰才能接收到来自父组件的数据。也就是说, 在这个例子中, 必须在子组件中用装饰器 `@Input` 修饰, 才能通过模板属性绑定的方式把数据流入到绑定的属性中, 从而在子组件中完成数据的接收。

拦截输入属性

父组件向子组件传递数据, 是通过属性绑定的方式将数据流向子组件的, 子组件可以拦截输入属性的数据并进行相应的处理。下面将介绍两种拦截处理方式, 即使用 `setter` 拦截输入属性和使用 `ngOnChanges` 监听数据变化。

使用 `setter` 拦截输入属性

`getter` 和 `setter` 通常配套使用, 用来对属性进行相关约束。它们提供了一些属性读写的封装, 使代码结构更清晰、更具可扩展性。`setter` 可以对属性进行再封装处理, 对复杂的内部逻辑通过访问权限控制来隔绝外部调用, 以避免外部的错误调用影响到内部的状态。同时也把内部复杂的逻辑结构封装成高度抽象可被简单调用的属性, 再通过 `getter` 返回要设置的属性值, 方便使用者调用。

上一节中讲到了通过输入属性来实现父子组件的数据交互。除此之外, 组件还可以通过属性 `setter` 来拦截来自父组件的数据源, 并对拦截到的数据进行处理, 使数据的输出更加合理、可控。在下面的例子中, 父组件将继续沿用前面的 `ListComponent` 组件, 仅对子组件 `ListItemComponent` 进行修改。示例代码如下:

```

@Component({
  selector: 'app-list-item',
  template: `
    <div>
      <label class="contact-name">{{ contactObj.name }}</label>
      <span class="contact-tel">{{ contactObj.telNum }}</span>
    </div>
  `,
})
export class ListItemComponent implements OnInit {
  _contact: object = {};
  @Input()
  set contactObj(contact: object) {
    this._contact.name = (contact.name && contact.name.trim()) || 'no name set';
    this._contact.telNum = contact.telNum || '000-000';
  }
  get contactObj() { return this._contact; }
}

```

这里通过 setter 的方式设置了一个 contactObj 属性对象，其作用是对通过 @Input 修饰符获取的数据 contact 进行二次处理，再通过 getter 返回这个 contactObj 对象。这样处理的作用是使得联系人不会出现 null 或者 undefined 的情况。getter 和 setter 其实是在该组件类的原型对象上设置了一个 contactObj 属性的。以下代码示例能加深读者对 getter 和 setter 的理解：

```

Object.defineProperty(ListItemComponent.prototype, "contactObj", {
  // ...
});

```

父组件 ListComponent 中的联系人通过属性绑定的方式将数据自上而下流入到其子组件中，contact 属性的数据在子组件 ListItemComponent 中通过 @Input 修饰符流入，被 setter 拦截器拦截，并对其数据进行再加工处理，使得数据处理的自由度更高、数据的展示也更符合预期。

使用 ngOnChanges 监听数据变化

ngOnChanges 用于及时响应 Angular 在属性绑定中发生的数据变化，该方法接收一个对象参数，包含当前值和变化前的值。ngOnChanges 在 ngOnInit 之前，或者当数据绑定的输入属性的值发生变化时会触发。ngOnChanges 是组件的生命周期钩子之一，关于更多的生命周期钩子将在本章 6.5 节进行讲解。

下面将通过例子来详细讲解使用 `ngOnChanges` 监听数据变化。在通讯录例子的详情页中，当父组件 `DetailComponent` 编辑联系人信息后，在子组件 `ChangeLogComponent`（为了方便说明，我们假设有 `ChangeLogComponent` 这个组件）中通过 `ngOnChanges` 来监听并处理数据的变化，将变化前后的信息通过日志的方式输出。请看父组件编辑联系人数据的例子，示例代码如下：

```
// detail.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'detail',
  template: `
    <a class="edit" (click)="editContact()">编辑</a>
    <change-log [contact]="detail"></change-log>
  `
})
export class DetailComponent implements OnInit {
  detail:any = {};
  // 完成联系人编辑修改
  editContact() {
    // ...
    this.detail = data; // 修改后的数据
  }
}
```



同样，已经在 `app.module.ts` 中通过 `@NgModule` 的元数据 `declarations` 将子组件 `ChangeLogComponent` 的实例引入到 `AppModule` 中，该模块中的任何组件都可以直接引用该组件。

我们想要的结果是通过单击“编辑”按钮来完成修改联系人数据操作，并将修改后的联系人数据变化情况通过子组件 `ChangeLogComponent` 进行输出，这里仅仅输出修改前后的数据对比。在单击“编辑”按钮后，调用 `editContact()` 方法完成联系人数据的修改，再通过 `<change-log>` 的属性绑定方式将数据绑定到 `contact` 属性，在子组件中完成接收工作，从而输出联系人修改前后的数据变化情况。

在讲解子组件处理变化日志之前，我们先来了解一下 `SimpleChanges` 类。它是 `Angular` 的一个基础类，用于处理数据的前后变化，其包含两个重要成员变量，分别是 `pre-`

viousValue 和 currentValue，其中 previousValue 用于获取变化前的数据；而 currentValue 用于获取变化后的数据。

子组件 ChangeLogComponent 负责把编辑联系人数据的前后变化通过日志的方式展示出来，示例代码如下：

```
// changelog.component.ts
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';

@Component({
  selector: 'change-log',
  template: `
    <h4>Change log:</h4>
    <ul>
      <li *ngFor="let change of changes">{{change}}</li>
    </ul>
  `,
})
export class ChangeLogComponent implements OnChanges {
  @Input() contact :any = {};
  changes: string[] = [];
  ngOnChanges(changes: {[propKey:string]: SimpleChanges}) {
    let log: string[] = [];
    for (let propName in changes) {
      let changedProp = changes[propName],
        from = JSON.stringify(changedProp.previousValue),
        to = JSON.stringify(changedProp.currentValue);
      log.push( `${propName} changed from ${from} to ${to}` );
    }
    this.changes.push(log.join(', '));
  }
}
```

在子组件中，通过 ngOnChanges 钩子方法来监测数据变化前后的情况。ngOnChanges 当且仅当组件输入数据变化时被调用，“输入数据”指的是通过 @Input 装饰器显式指定的那些数据。这个例子中的 previousValue 是修改前联系人的数据，currentValue 是修改后联系人的数据，最终结果是每次单击父组件的“编辑”按钮，都会子组件 ChangeLogComponent 中记录数据的变化并输出到页面上。

6.3.3 子组件向父组件传递数据

使用事件传递是子组件向父组件传递数据最常用的方式。子组件需要实例化一个用来订阅和触发自定义事件的 `EventEmitter` 类，这个实例对象是一个由装饰器 `@Output` 修饰的输出属性，当有用户操作行为发生时该事件会被触发，父组件则通过事件绑定的方式来订阅来自子组件触发的事件，即子组件触发的具体事件会被其父组件订阅到。

下面将通过事件传递的方式来实现联系人详情页中收藏联系人的例子。我们在收藏页面组件 `CollectionComponent` 中添加一个子组件 `ContactCollectComponent`，该子组件包含“收藏”按钮。单击“收藏”按钮后将完成联系人的收藏操作，即在子组件 `ContactCollectComponent` 中通过数据绑定的方式实现了单击收藏的功能，具体的收藏操作统一在父组件中实现。我们先来看父组件 `CollectionComponent`，示例代码如下：

```
// collection.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'collection',
  template: `
    <contact-collect [contact]="detail" (onCollect)="collectTheContact($event)"></
      contact-collect>
  `
})
export class CollectionComponent implements OnInit {
  detail:any = {};
  collectTheContact() {
    this.detail.collection == 0 ? this.detail.collection = 1 : this.detail.
      collection = 0;
  }
}
```



在 `app.module.ts` 中通过 `@NgModule` 的元数据 `declarations` 将子组件 `ContactCollectComponent` 的实例引入到 `AppModule` 中，在该模块的任何组件中都可以直接引用该组件，下同。

父组件 `CollectionComponent` 通过绑定自定义事件 `onCollect` 订阅来自子组件触发的事件。当有来自子组件对应的事件被触发时，在父组件中能够监听到该事件，以此来完

成收藏功能，具体的收藏操作是在父组件的 `collectTheContact()` 方法中实现的。

下面将介绍子组件 `ContactCollectComponent` 的具体实现。子组件绑定一个点击事件，用户执行收藏操作来完成联系人的收藏。当“收藏”按钮被单击后，将会触发父组件的自定义事件 `onCollect`，从而在父组件中完成联系人的收藏。示例代码如下：

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'contact-collect',
  template: `
    <i [ngClass]="{collected: contact.collection}" (click)="collectTheContact()">收藏</i>
  `
})
export class ContactCollectComponent {
  @Input() contact:any = {};
  @Output() onCollect = new EventEmitter<boolean>();
  collectTheContact(){
    this.onCollect.emit();
  }
}
```

在上面的例子中，单击“收藏”按钮后将触发自定义事件 `onCollect = new EventEmitter<boolean>()`，通过输出属性 `@Output` 将数据流向父组件，在父组件中完成事件的监听，以此来实现从子组件到父组件的数据交互。要完成这样的数据通信主要依赖 `@Output`，它声明事件绑定的输出特性，当输出属性发出一个事件后，在模板中绑定的对应事件处理句柄（Event Handler）将会被调用。

6.3.4 其他组件交互方式

父子组件间数据传递的实现还有其他方式，这里主要介绍以下两种：

- 父组件通过局部变量获取子组件的引用。
- 父组件使用 `@ViewChild` 获取子组件的引用。

通过局部变量实现数据交互

前面章节中实现的父子组件间数据交互，都是通过输入、输出属性绑定的方式来实现数据双向流动的。但父组件仅仅是将数据源流向下级子组件，它不拥有读取子组件的

相关成员变量和方法的权限，因此也不能调用子组件的相关成员变量和方法。

在 Angular 中，“模板局部变量”可以帮助我们获取子组件的实例引用。通过创建模板局部变量的方式来实现父组件与子组件的数据交互，即在父组件的模板中为子组件创建一个局部变量，那么父组件就可以通过这个局部变量来获得读取子组件公共成员变量和方法的权限。模板局部变量的作用域范围仅存在于定义该模板局部变量的子组件中。还是用通讯录收藏功能的例子来进行说明，可以对父组件 `CollectionComponent` 进行修改，示例代码如下：

```
import { Component } from '@angular/core';
@Component({
  selector: 'collection',
  template: `
    <contact-collect (click)="collect.collectTheContact()" #collect></contact-collect>
  `,
})
export class CollectionComponent {}
```

在上面的例子中，通过单击“收藏”按钮来实现联系人收藏功能，具体功能在子组件中实现，在父组件模板中的 `<contact-collect>` 子组件标签上绑定一个局部变量 `collect`（以“#”号标记），以此来获取子组件类的实例对象。

在子组件 `ContactCollectComponent` 中实现收藏联系人的功能，示例代码如下：

```
import { Component } from '@angular/core';
@Component({
  selector: 'contact-collect',
  template: `
    <i [ngClass]="{collect: detail.collection}">收藏</i>
  `,
})
export class ContactCollectComponent {
  detail:any = {};
  collectTheContact() {
    this.detail.collection == 0 ? this.detail.collection = 1 : this.detail.collection = 0;
  }
}
```

子组件在 `collectTheContact()` 方法中实现了收藏功能，但是在父组件中是没有调用 `collectTheContact` 的权限的，同样也不可能获取到这个联系人的 `detail` 属性。这里通过定义模板局部变量 `collect` 来实现我们想要的功能，即在标签元素 `<contact-collect>` 中放置一个局部变量 `collect`，用来获取子组件的实例引用，以此来访问子组件中的成员变量和方法。

以上就是父子组件通过模板局部变量完成数据交互的实现方式。

使用 `@ViewChild` 实现数据交互

使用模板局部变量实现父组件调用子组件的成员变量和方法，看上去简单、容易。但是模板变量只能在模板中使用，而不能直接在父组件类里使用，这有一定的局限性。接下来将介绍另一种更优雅的数据传递方式——使用 `@ViewChild`。

当父组件需要获取到子组件中变量或者方法的读写权限时，可以通过 `@ViewChild` 注入的方式来实现。组件中元数据 `ViewChild` 的作用是声明对子组件元素的实例引用，它提供了一个参数来选择将要引用的组件元素，这个参数可以是一个类的实例，也可以是一个字符串，它们实现的功能是一样的，只是表现形式不同。具体如下：

- 参数为类实例，表示父组件将绑定一个指令或者子组件实例。
- 参数为字符串类型，表示将起到选择器的作用，即相当于在父组件中绑定一个模板局部变量，获取到子组件的一个实例对象的引用。

组件中元数据 `ViewChild` 的参数为字符串的实现方式，和之前介绍的绑定模板局部变量是一样的，这里将重点讲解参数为类实例的情景。下面继续使用通讯录收藏功能的例子，要实现一样的功能，其中子组件需要保留之前的代码，我们对父组件 `ContactCollectComponent` 的代码进行修改，示例代码如下：

```
import { Component, AfterViewInit, ViewChild } from '@angular/core';
@Component({
  selector: 'collection',
  template: `
    <contact-collect (click)="collectTheContact()"></contact-collect>
  `
})
export class CollectionComponent {
  @ViewChild(ContactCollectComponent) contactCollect: ContactCollectComponent;

  ngAfterViewInit() {
```

```
// ...  
}  
  
collectTheContact() {  
  this.contactCollect.collectTheContact();  
}  
}
```

这种实现方式把单击“收藏”按钮事件绑定到父组件中，同时在父组件类中调用子组件的 `collectTheContact()` 方法。即通过 `@ViewChild` 装饰器将 `ContactCollectComponent` 子组件注入进来，并赋值给 `contactCollect` 变量，这个变量就是对子组件实例的引用，从而达到了从子组件到父组件进行数据交互的目的。

在父组件中获取子组件的实例比较容易；反过来，在子组件中获取父组件的实例就相对麻烦些，这种场景可通过依赖注入机制间接找到父组件的实例，从而获取到父组件的实例对象。但这里不再继续展开，具体请参阅第 10 章。

6.4 组件内容嵌入

内容嵌入（`ng-content`）是组件的一个高级功能特性，使用组件的内容嵌入特性能很好地扩充组件的功能，方便代码复用。



接触过 AngularJS 1.x 的读者，相信对组件的内容嵌入不会陌生，它和 AngularJS 1.x 中指令的 `transclude` 属性非常类似。

内容嵌入通常用来创建可复用的组件，典型的例子是模态对话框和导航栏。在开发 Web 应用的时候，模态对话框和导航栏是使用非常频繁的 UI 组件，而内容嵌入特性提供了一种复用的方式，使得这些组件具有一致的样式，但内容又可以自定义。例如在通讯录例子中，可以看到无论哪一页，头部（Header）都是一样的样式，如图 6-4 所示。

在这种情况下，就可以考虑把它改造成可复用组件。这里以一个简化的例子来说明这种场景。定义一个 `NgContentExampleComponent` 组件，然后把它设置成灰色的背景，里面的内容可动态变化，如图 6-5 所示。

`NgContentExampleComponent` 组件的示例代码如下：

```
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'example-content',
  template: `
    <div>
      <h4>ng-content 示例</h4>
      <div style="background-color: gray; padding: 5px; margin: 2px">
        <ng-content select="header"></ng-content>
      </div>
    </div>
  `,
})
class NgContentExampleComponent {}
```



图 6-4 通讯录例子中不同页面的头部

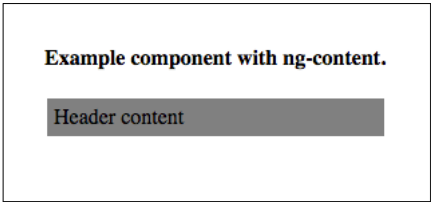


图 6-5 自定义 Header 内容的例子

如上，在模板中使用了 `<ng-content>` 标签，这个标签就是用来渲染组件嵌入内容的。在 `<ng-content>` 中有一个 `select="header"`，用于匹配内容，并填充到 `ng-content` 中。

接着再定义一个简单的根组件来使用它，示例代码如下：

```
import { Component } from '@angular/core';

@Component({
```

```

    selector: 'app',
    template: `
      <example-content>
        <header>Header content</header>
        <!-- 将自定义的内容放到 example-content 标签之间 -->
      </example-content>
    `
  })

```

```
export class NgContentAppComponent {}
```

最后组件的 DOM 树会被 Angular 渲染成：

```

<app>
  <example-content>
    <div>
      <h4>ng-content 示例</h4>
      <div style="background-color: gray; padding: 5px; margin: 2px">
        <header>Header content</header>
      </div>
    </div>
  </example-content>
</app>

```

注意到 `<example-content>` 标签之间的内容，也就是 `<header>Header content</header>`，被填充到 `ng-content`，而 `NgContentExampleComponent` 组件模板中的其他元素没有受到影响。那么嵌入内容是如何匹配显示的呢？上文提及了 `select="header"`，`select` 属性是一个选择器，它与 CSS 选择器的作用是类似的，表示匹配 `<example-content>` 标签之间的第一个 `<header>` 标签，并将其填充到相应的 `ng-content` 中。

另外，还可以同时使用多个嵌入内容。下面修改 `NgContentExampleComponent` 组件的代码，通过标签选择器、类选择器、属性选择器来指定多个 `ng-content`。示例代码如下：

```

import { Component } from '@angular/core';

@Component({
  selector: 'example-content',
  template: `
    <div>
      <h4>Example component with ng-content.</h4>
      <div style="background-color: green; padding: 5px; margin: 2px">

```



```

        <ng-content select="header"></ng-content>
    </div>
    <div style="background-color: gray; padding: 5px; margin: 2px">
        <ng-content select=".class-select"></ng-content>
    </div>
    <div style="background-color: blue; padding: 5px; margin: 2px">
        <ng-content select="[name=footer]"></ng-content>
    </div>
</div>
、
})
export class NgContentExampleComponent {}

```

然后修改 `NgContentAppComponent` 组件的代码来使用多个嵌入内容。示例代码如下：

```

import { Component } from '@angular/core';

@Component({
  selector: 'app',
  template: `
    <example-content>
      <header>Header content</header>
      <div class="class-select">
        div with .class-select
      </div>
      <div name="footer">Footer content</div>
    </example-content>
  `
})
export class NgContentAppComponent {}

```

最终的效果如图 6-6 所示。

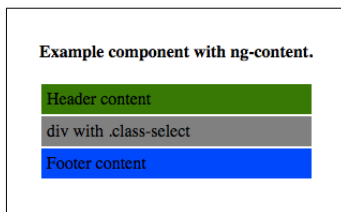


图 6-6 多个嵌入内容

6.5 组件生命周期

6.5.1 概述

组件的生命周期由 Angular 内部管理，从组件的创建、渲染，到数据变动事件的触发，再到组件从 DOM 中移除，Angular 提供了一系列的钩子。这些钩子可以让开发者很方便地在这些事件被触发时，执行相应的回调函数。

6.5.2 生命周期钩子

开发者可以实现一个或者多个生命周期钩子（接口），从而在生命周期的各阶段做出适当的处理。这些钩子接口包含在 @angular/core 中。每个接口都对应一个名为“ng + 接口名”的方法，例如 OnInit 接口有一个叫 ngOnInit 的钩子方法。示例代码如下：

```
class ExampleInitHook implements OnInit {  
  constructor() {}  
  ngOnInit() {  
    console.log('OnInit');  
  }  
}
```

以下是组件常用的生命周期钩子方法，Angular 会按以下顺序依次调用钩子方法：

- ngOnChanges
- ngOnInit
- ngDoCheck
- ngAfterContentInit
- ngAfterContentChecked
- ngAfterViewInit
- ngAfterViewChecked
- ngOnDestroy

除此之外，有的组件还提供了自己特有的生命周期钩子，例如路由有 routerOnActivate 钩子。接下来将介绍这些钩子的相关信息。

ngOnChanges

上文介绍组件交互时提到过这个钩子，它是用来响应组件输入值发生变化时触发的事件。该方法接收一个 SimpleChanges 对象，包含当前值和变化前的值。该方法在

ngOnInit 之前，或者当数据绑定输入属性的值发生变化时会被触发。

在 AngularJS 1.x 中，如果想要监听数据的变化，则需要设置 `$scope.$watch`，然后在每次 `digest` 循环里判断数据是否有改变。在 Angular 中，`ngOnChanges` 钩子把这个过程变得更加简单。只要在组件里定义了 `ngOnChanges` 方法，当输入数据发生变化时该方法就会被自动调用。

需要注意的是，`ngOnChanges` 当且仅当组件输入数据变化时被调用，这里的“输入数据”指的是通过 `@Input` 装饰器显式指定的那些变量。

ngOnInit

`ngOnInit` 钩子用于数据绑定输入属性之后初始化组件。该钩子方法会在第一次 `ngOnChanges` 执行后被调用。

使用 `ngOnInit` 有以下两个重要原因：

- 组件构造后不久就要进行复杂的初始化。
- 需要在输入属性设置完成之后再构建组件。

在组件中，经常会使用 `ngOnInit` 获取数据。为什么不在组件构造函数中获取数据呢？首先，构造函数做的事，例如成员变量初始化，应该尽可能简单，这对于有经验的开发人员来说，已经是一种共识。另外，这对于 Angular 自动化测试的一些场景也有非常重要的作用，把与业务相关的初始化代码放到 `ngOnInit` 里可以很容易进行 Hook 操作，而构造函数不能被显式调用，因此无法进行 Hook 操作。

ngDoCheck

`ngDoCheck` 用于变化监测，该钩子方法会在每次变化监测发生时被调用。

在每一个变化监测周期内，不管数据值是否发生了变化，`ngDoCheck` 都会被调用。但要慎用这个钩子方法，例如鼠标移动时会触发 `mousemove` 事件，此时变化监测会被频繁触发，随之 `ngDoCheck` 也会被频繁调用。因此，在 `ngDoCheck` 方法中不能写一些复杂的代码，否则性能就会受到影响。

在绝大多数情况下，`ngDoCheck` 和 `ngOnChanges` 不应该一起使用。`ngOnChanges` 能做的事情，`ngDoCheck` 也能做，而且 `ngDoCheck` 监测的粒度更小，可以完成更灵活的变化监测逻辑。

ngAfterContentInit

在组件中使用 `<ng-content>` 将外部内容嵌入到组件视图后就会调用 `ngAfterContentInit`，它在第一次 `ngDoCheck` 执行后被调用，且只执行一次。

ngAfterContentChecked

在组件使用了 `<ng-content>` 自定义内容的情况下，Angular 在这些外部内容嵌入到组件视图后，或者每次变化监测时都会调用 `ngAfterContentChecked`。

ngAfterViewInit

`ngAfterViewInit` 会在 Angular 创建了组件的视图及其子视图之后被调用。

ngAfterViewChecked

`ngAfterViewChecked` 在 Angular 创建了组件的视图及其子组件视图之后被调用一次，并且在每次子组件变化监测时也会被调用。

ngOnDestroy

`ngOnDestroy` 在销毁指令/组件之前被触发。那些不会被垃圾回收器自动回收的资源（比如已订阅的观察者事件、绑定过的 DOM 事件、通过 `setTimeout` 或 `setInterval` 设置过的计时器，等等）都应当在 `ngOnDestroy` 中手动销毁，从而避免发生内存泄漏等问题。

6.6 变化监测

Angular 提供了数据绑定的功能（在“模板”章节中会详细介绍）。所谓数据绑定就是将组件类的数据和页面的 DOM 元素关联起来。当数据发生变化时，Angular 能够监测到这些变化，并对其所绑定的 DOM 元素进行相应的更新，反之亦然。

异步事件的发生会导致组件中数据的变化，但 Angular 并不捕获对象的变动，它采用的是在适当的时机来检验对象的值是否被改动。这个时机是由 `NgZone` 这个服务掌控的，它获取到了整个应用的执行上下文，能够对相关的异步事件发生、完成或者异常等进行捕获，然后驱动 Angular 的变化监测机制执行。下面将详细介绍各个功能模块的细节。

6.6.1 数据变化的源头

在应用程序中，大致有三种引起数据变化的应用场景。第一种是用户的行为操作，即页面操作所引发的用户事件，如 `click`、`change`、`hover` 等，以此对用户的操作做出响应；第二种是前后端的数据交互，比如从后端服务拉取页面所需要的接口数据，如 `XMLHttpRequest` / `WebSocket` 等，这是一个异步的过程，获取到一份较之前可能有变化的数据；第三种是各类定时任务，即在某个延时后再来响应对应的操作，从而对页面数据做出改变，如 `setTimeout`、`setInterval`、`requestAnimationFrame` 等，它们都是延后到某个时间才触发相应的操作的。这些能引起数据变化的异步处理总结如图 6-7 所示。

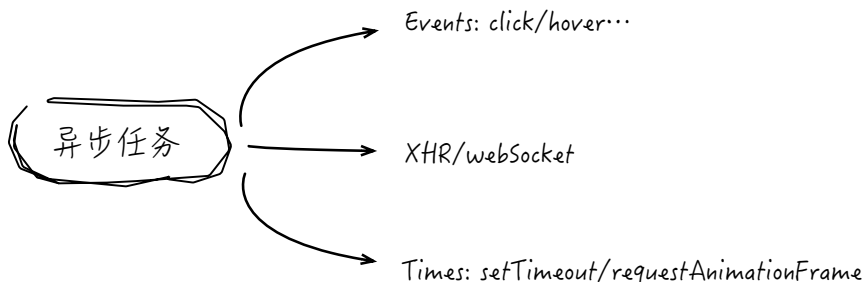


图 6-7 异步处理总结

以上三种可能导致数据变化的场景有一个共同特征，即它们都是异步的处理，使用异步回调函数句柄来处理相关的数据操作。因此，任意一个异步操作，都有可能数据层面发生改变，这会导致应用程序状态被改变。如果可以在每一个异步回调函数执行结束后，通知 `Angular` 内核进行变化监测，那么任何数据的更改都可以在视图层实时地反映出来。

下面举个简单的例子进行说明，可以在组件的模板元数据 `template` 中添加如下代码片段：

```
// ...  
<i [ngClass]="{collect: detail.collection}" (click)="collectTheContact()">收藏</i>  
// ...
```

该例子是通讯录收藏联系人的一个功能实现。当用户点击“收藏”按钮后，这个操作会改变组件里的收藏数据对象 `detail.collection`，并通知 `Angular` 来检查数据的变化，在视图层做出相应的改变，如改变 `DOM` 元素样式等。

6.6.2 变动通知机制

当有异步事件被触发时，Angular 将进行变化监测，任何数据的变更都可以被实时地在视图层反映出来。实际上 Angular 自身并没有实现捕获异步事件的功能，而是引入了 Zone.js 这个强大的异步事件库来解决的。

Zone.js 是一个特殊的补丁包，它主要通过猴子补丁的方式来实现异步事件的捕获，即运行时重写所有的浏览器异步事件 API（如 `setTimeout`、`XHR` 等）。所以，Zone.js 就可感知这些异步事件的执行及其上下文环境，一旦有异步事件发生时，即可在适当的时机触发 Angular 变化监测。

Zone.js 记录这些异步事件的上下文环境是在一个名为“Zone”的对象里完成的，每个 Zone 对象都保存了异步事件的执行信息，以及暴露了异步事件执行的生命周期钩子（如异步事件执行前或执行出错等）。Zone.js 初始化时会生成一个初始的 Zone 对象，称为 Root Zone。Zone 对象可以通过 `fork` 操作生成子 Zone，异步事件都是挂靠在某个 Zone（Root Zone 或子 Zone 都可以）之下运行的，而不同的 Zone 之间有一定的隔离性。

需要明确的一点是，在默认情况下，整个 Angular 应用是运行在某个子 Zone 之下的，这个子 Zone 就称为 Angular Zone。这意味着 Angular 应用中触发的所有异步事件都可被 Angular Zone 感知（可在异步事件执行后进行一些必要操作，如触发变化监测等），而 Root Zone 或其他子 Zone 是感知不到 Angular 的异步事件执行的（即不会触发变化监测）。了解这点很重要，后文会述及。

为了管理 Root Zone 和子 Zone，Angular 为 Zone.js 又封装了一层，称为 `NgZone`。`NgZone` 本身是一个服务（关于 Angular 服务后面会讲到）。除管理各个 Zone 对象外，`NgZone` 还封装了一些支持 Angular 运行的友好事件，当有异步任务发生、完成或者抛出异常时，都可以监听对应的事件并进行捕获处理。这些事件列举如下。

- `onUnstable`：当代码进入 Angular 的执行环境时被触发，在 VM（JavaScript Virtual Machine）中最先被触发，即通常在异步任务执行前被触发。
- `onMicrotaskEmpty`：当 VM 中 `Microtasks` 队列为空时被触发，用于提示 Angular 执行变化监测。
- `onStable`：当最后一个 `onMicrotaskEmpty` 已经执行完成并且 `Microtasks` 队列为空时被触发，这个事件仅仅被触发一次，即通常在异步事件执行完成且变化监测已经执行完成时被触发。
- `onError`：当有错误异常抛出时被触发。

上述这些事件在日常开发中使用的场景比较少，开发者在大多数时候更关心的是触发变化监测的频率。在默认情况下，几乎每一次异步事件触发都会执行变化监测，有些时候这并不是开发者所需要的，如 `mousemove`、`scroll` 事件，如果这些事件频繁地触发变化监测，就很容易导致页面卡顿。因此，`NgZone` 提供了 `run()` 和 `runOutsideAngular()` 两个方法帮助开发者控制变化监测是否执行。

前面提到过，Angular 应用默认运行在 Angular Zone 下，所以只有在 Angular Zone 下触发的事件才会触发变化监测，而在 Root Zone 下触发的事件不会触发变化监测。`runOutsideAngular()` 方法让 Angular 应用绑定的事件逃离 Angular Zone 执行，即在 Root Zone 下执行。需要注意的是，一旦执行了 `runOutsideAngular()`，在该回调函数里后续新绑定的异步事件也会一直在 Root Zone 下执行，直到遇到 `run()` 方法。`run()` 方法对 `runOutsideAngular()` 进行还原操作，将这些逃离的异步事件重新挂靠到 Angular Zone 下执行。

利用这两个 API，开发者可以在某些场景（如拖放）中提升 Angular 应用的性能。如果某个元素需要响应用户的拖放，则通常给这个元素加上 `mousedown`、`mousemove`、`mouseup` 这三个鼠标事件。示例代码如下：

```
<div
  (mousedown)="doMouseDown($event)"
  (mousemove)="doMouseMove($event)"
  (mouseup)="doMouseUp($event)">
</div>
```

`mousemove` 事件会频繁触发变化监测执行，容易触达性能瓶颈。所以，更好的做法是让 `mousemove` 事件不触发变化监测，可以通过 `runOutsideAngular()` 方法来实现。示例代码如下：

```
doMouseDown(event) {
  // ...
  this.zone.runOutsideAngular(() => {
    window.document.addEventListener('mousemove', this.doMouseMove.bind(this));
  });
}
```



上面代码中的 `this.zone` 是通过依赖注入方式引入 `NgZone` 实例的，详情可在第 10 章中查看。

通过在 `runOutsideAngular()` 回调函数里使用原始的 DOM API 绑定 `mousemove` 事件,使得 `mousemove` 运行在 `Root Zone` 下,不再触发变化监测。由于 `mousemove` 是通过 DOM API 方式绑定的,所以还需要在 `mouseup` 的回调函数里解绑 `mousemove` 事件,并需要原来的模板移除 `mousemove` 事件绑定。示例代码如下:

```
<div
  (mousedown)="doMouseDown($event)"
  (mouseup)="doMouseUp($event)">
</div>
```

现在梳理一下 Angular 的变动通知机制。在默认情况下,所有的异步事件都是在 `Angular Zone` 下执行的,每当有异步事件触发时,无论是否导致了数据变化,都会执行变化监测逻辑。

变化监测的入口函数定义在 `ApplicationRef` 这个服务类里,函数名为 `tick()`,精简的源码示例如下:

// 精简的源码版本

```
class ApplicationRef {
  changeDetectorRefs: ChangeDetectorRef[] = [];
  constructor(private zone: NgZone) {
    this.zone.onMicrotaskEmpty.subscribe(() => this.zone.run(
      () => this.tick()
    ));
  }
  tick() {
    this.changeDetectorRefs.forEach((ref) => ref.detectChanges());
  }
}
```

Angular 变化监测是以组件为执行单元的,在默认的变化监测策略里,每一次变化监测都从根组件开始,以深度优先的原则向子组件遍历,直至整棵组件树的所有组件都执行一次变化监测。有些时候我们的应用不需要所有的组件都执行一次变化监测,所以可以通过一些策略调整来提升变化监测的性能。接下来就将介绍变化监测的响应策略。

6.6.3 变化监测的响应处理

我们已经知道了触发 Angular 执行变化监测的时机及变动通知机制,接下来看一下 Angular 内部是如何对数据变动进行变化监测的。

变化监测的处理机制

Angular 应用由大大小小的组件组成，这些有相互依赖关系的组件组成了一棵线性的组件树。此外，每一个组件都有自己的变化监测器，由此组成了一棵变化监测树。变化监测树的数据是由上到下单向流动的，这是因为变化监测的执行总是由根组件开始，从上到下监测每一个组件的变化。单向的数据流让人清晰地了解视图中数据的来源，明白数据的变化是由哪个组件引起的。

为了更清晰地理解这些概念，下面将结合通讯录中联系人的例子来说明，该组件树如图 6-8 所示。

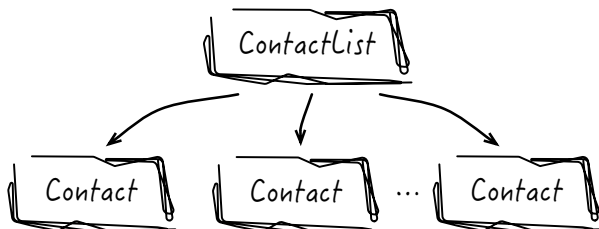


图 6-8 组件树

在联系人列表中，由 `ListComponent` 组件获取到联系人列表数据，在其子组件 `ListItemComponent` 中展示具体的联系人数据。如前面所讲，当一个异步事件发生并导致其中的组件数据发生改变时，在组件中绑定的相关处理事件将会被触发，事件句柄处理完成相关逻辑之后，`NgZone` 将会执行对应的钩子方法并通知 Angular 执行一次变化监测。

不同组件间的相互关联形成了组件树，组件树中的每个组件都有其对应的变化监测器，使得每个组件的变化监测相互独立，可以更灵活地控制变化监测的执行或暂停等，对提升性能具有重要的意义。

那么在这棵组件树中，变化监测器是如何工作的呢？当组件中数据有变动时，`NgZone` 通过钩子捕获到变化并通知 Angular 执行变化监测。变化监测是单向线性的，即从根组件开始，依次触发各个子组件的变化监测器来完成变化的对比工作。在每个组件的执行环境中，Angular 都会创建一个变化监测类的实例，该实例能准确地记录每个组件的数据模型，并以此作为下一轮变化监测的参考标准。

在默认情况下，任何一个组件模型中的数据变化都会导致整棵组件树的变化监测，但其实很多组件的输入属性是没有变化的，因此没必要对这样的组件进行变化监测操作。减少不必要的监测操作可以提升应用程序的性能。如果想学习更多的关于变化监测优化的内容，首先要深入理解变化监测类（`ChangeDetectorRef`）。

变化监测类

Angular 在整个运行期间会为每一个组件创建变化监测类的实例，该实例提供了相关的方法来手动管理变化监测。正如前文所提到的，当数据发生变化时，Angular 会从根组件到子组件来监测每个组件是否发生了变化。Angular 并不知道哪个组件发生了变化，但开发者知道，所以可以给这个组件做标记，以此来通知 Angular 仅仅监测这个组件所在路径上的组件即可。

变化监测类 `ChangeDetectorRef` 提供的主要接口如下：

```
class ChangeDetectorRef {  
  // ...  
  markForCheck() : void  
  detach() : void  
  detectChanges() : void  
  reattach() : void  
}
```

上述各个接口的功能介绍如下。

- `markForCheck()`：把从根组件到该组件之间的这条路径标记起来，通知 Angular 在下次触发变化监测时必须检查这条路径上的组件。
- `detach()`：从变化监测树中分离变化监测器，该组件的变化监测器将不再执行变化监测，除非再次手动执行 `reattach()` 方法。
- `detectChanges()`：手动触发执行该组件到各个子组件的一次变化监测。
- `reattach()`：把分离的变化监测器重新安装上，使得该组件及其子组件都能执行变化监测。

下面将通过例子来说明其中一些方法的使用场景。假设通讯录中联系人的数据时刻在变化着，而产品需求是不需要实时地根据变化来展示数据，那么为了性能的考虑，可以设置在一定时间范围内来执行变化监测。这里我们通过 `detach()` 和 `detectChanges()` 方法配合使用来实现性能的优化。示例代码如下：

```
// ...  
@Component({  
  selector: 'list',  
  template: `  
    <ul class="list">  
      <li *ngFor="let contact of contacts">  
        <list-item [contact]="contact"></list-item>  
      </li>  
    </ul>  
  `
```

```

    </li>
  </ul>
  、
})
export class ListComponent implements OnInit {
  contacts:any = {};

  constructor(
    // ...
    private cd: ChangeDetectorRef
  ) {
    cd.detach();
    // 定时手动执行变化监测
    setInterval(() => {
      this.cd.detectChanges();
    }, 5000);
  }

  ngOnInit() {
    this.getContacts();
  }

  getContacts() {
    this.contacts = data; // data 是获取到的联系人列表数据
  }
  // ...
}

```

`detach()` 方法用于从当前组件中分离变化监测器，在需要变化监测的时候再通过 `detectChanges()` 手动触发变化监测。在列表数据变化频繁的情况下，这样的处理方式能起到很大的优化作用。

变化监测策略

在 Angular 中，每个组件都包含一些元数据，其中一些是可选的，例如本节要介绍的 `changeDetection`，它的作用是让开发者定义每个组件的变化监测策略。在使用该功能前，需要先导入 `ChangeDetectionStrategy` 对象。示例代码片段如下：

```

import { Component, ChangeDetectionStrategy } from '@angular/core';
// ...

```

```
@Component({
  // ...
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ContactComponent {
  // ...
}
```

ChangeDetectionStrategy 枚举类型值有两种，分别是 Default 和 OnPush。当值为 Default 时，组件的每次变化监测都会检查其内部的所有数据（引用对象也会被深度遍历），以此得到前后的数据变化情况；而当值为 OnPush 时，组件的变化监测只检查输入属性（即 @Input() 修饰的变量）的值是否发生变化，当这个值为引用类型（如 Object、Array 等）时，则只对比该值的引用。

显然，OnPush 策略相比 Default 降低了变化监测的复杂度，很好地提升了变化监测的性能。如果子组件的更新只依赖输入属性的值，那么在子组件上使用 OnPush 策略是一个很好的选择。但是 OnPush 策略只对比值的“引用”，这在某些场景下可能会得不到预期的效果。举个例子：子组件通过输入属性获取了父组件的一个 Object 值，如 {a:1,b:2}，父组件修改了该对象内的值，例如改成 {a:11,b:2}，这时候对象的引用没有发生变化，因此子组件的变化监测并不能感知到对象已变化。在大多数情况下，这不是我们想要的效果。如果希望子组件也能正常更新数据，解决办法有两个：

- 修改变化监测策略为 Default，但这样会牺牲性能。
- 使用 Immutable 对象来传值，这是比较推荐的做法。

使用 Immutable 对象可以确保当对象值的引用地址不变时，对象内部的值或结构也会保持不变；反之，当对象内部发生变化时，对象的引用必然会发生改变。这个特性满足了在 OnPush 策略下，变化监测只对比引用地址即可知道数据是否发生了变化。

下面举个简单的例子来进行说明。首先来看子组件，示例代码如下：

```
import { Component, Input, ChangeDetectionStrategy } from '@angular/core';

@Component({
  selector: 'app-list-item',
  template: `
    <div>
      <label>{{ contact.get('name') }}</label>
      <span>{{ contact.get('telNum') }}</span>
    </div>
```

```

    `,
    changeDetection: ChangeDetectionStrategy.OnPush
  })
  export class ListItemComponent {
    @Input() contact:any = {};
    // ...
  }

```

从上述代码可以看出，子组件的数据更新只依赖输入属性 `contact` 的值，所以使用 `OnPush` 策略可以满足需求。再看父组件，示例代码如下：

```

import { Component } from '@angular/core';
import Immutable from 'immutable';

@Component({
  // ...
  template: `
    <app-list-item [contact]="contactItem"></app-list-item>
    <button (click)="doUpdate()">更新</button>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ListComponent {
  contactItem: any;
  constructor() {
    this.contactItem = Immutable.map({
      name: '张三',
      telNum: '12345678'
    })
  }

  doUpdate() {
    this.contactItem = this.contactItem.set('telNum', '87654321');
  }
}

```

父组件引入了 `Immutable` 工具库，并且把 `contactItem` 包装成 `Immutable` 对象。当点击“更新”按钮时，`contactItem` 被赋值为新的对象引用，子组件监测到 `contactItem` 的引用变化，从而触发数据更新。

6.7 扩展阅读

6.7.1 元数据一览表

组件的元数据一览表，如表 6-1 所示。

表 6-1 组件的元数据一览表

名称	类型	作用
selector	string	自定义组件的标签，用于匹配元素
inputs	string[]	指定组件的输入属性
outputs	string[]	指定组件的输出属性
host	{{key: string}: string;}	指定指令/组件的事件、动作和属性等
providers	any[]	指定该组件及其所有子组件（含 ContentChildren）可用的服务（依赖注入）
exportAs	string	给指令分配一个变量，使得可以在模板中调用
moduleId	string	包含该组件模块的 id，它被用于解析模板和样式的相对路径
queries	{{key: string}: any;}	设置需要被注入到组件的查询
viewProviders	any[]	指定该组件及其所有子组件（不含 ContentChildren）可用的服务
changeDetection	ChangeDetectionStrategy	指定使用的变化监测策略
templateUrl	string	指定组件模板所在的路径
template	string	指定组件的内联模板
styleUrls	string[]	指定组件引用的外部样式文件
styles	string[]	指定组件使用的内联样式
animations	AnimationEntryMetadata[]	设置 Angular 动画
encapsulation	ViewEncapsulation	设置组件的视图包装选项
interpolation	[string, string]	设置自定义插值标记，默认是双大括号“{{ }}"
preserveWhitespaces	Boolean	设置是否保留模板里制表符、换行符和空白，当前默认为 true

6.7.2 元数据说明

在本章的内容中已对 `selector`、`inputs`、`outputs`、`providers`、`templateUrl`、`template`、`styleUrls`、`styles`、`changeDetection` 等元数据做了不同程度的讲解，以下将讲解表 6-1 中未提及的大部分元数据。

host

`host` 是一个功能非常强大的元数据，它主要用在指令中。通过 `host` 可以指定此指令/组件的事件、动作和属性等。因为组件是指令的一种类型，所以在指令的 `@Directive` 装饰器中的配置也同样适用于组件。

exportAs

此属性主要在指令中使用，其作用是将指令分配给一个变量，使用这个名称就可以在模板中调用指令。同样，在 `@Directive` 中的配置也适用于组件。

moduleId

包含该组件模块的 `id`，它被用于解析模板和样式的相对路径。在 Dart 中，它可以被自动确定，并不需要手动设置。在 CommonJS 中，它总是被设置为 `module.id`。在这种情况下，如果 CSS、HTML、TypeScript 文件在同一个目录如 `app` 下，则可以去除基准路径如 `“app/”`。示例代码如下：

```
@Component({
  moduleId: module.id,
  templateUrl: 'some.component.html',
  styleUrls: ['some.component.css']
})
```

选用 Angular CLI 方案的开发者，不需要手动添加 `moduleId` 配置，直接采用以 `“./”` 开头的相对路径写法即可。示例代码如下：

```
@Component({
  templateUrl: './some.component.html',
  styleUrls: ['./some.component.css']
})
```

queries

设置需要被注入到组件的查询。在组件中主要有两种查询,即视图查询 (ViewChild) 和内容查询 (ContentChild), 它们分别会在 ngAfterViewInit 和 ngAfterContentInit 回调函数被调用之前设置。

通过 queries 属性, 开发者可以获取模板的元素, 甚至是子组件的引用。下面的例子展示了获取子组件的实例引用, 并调用子组件的 doSomething() 方法。示例代码如下:

```
import ChildComponent from './child.component';
@Component({
  selector: 'parent',
  template: `
    <div>
      <child />
    </div>
  `,
  queries: {
    child: new ViewChild(ChildComponent)
  }
})
export class ParentComponent implements AfterViewInit {
  child: ChildComponent;

  ngAfterViewInit() {
    this.child.doSomething();
  }
}
```

上面的代码可用注解的方式重写。示例代码如下:

```
@Component({
  selector: 'parent',
  template: `
    <div>
      <child />
    </div>
  `,
})
export class ParentComponent implements AfterViewInit {
  // 使用注解的方式
```



```

@ViewChild(ChildComponent)
child: ChildComponent;

ngAfterViewInit() {
  this.child.doSomething();
}
}

```

以上便是视图查询的用法。简而言之，就是借助于视图查询和 Angular 提供的 API 获取在组件模板上定义的节点引用。那么内容查询又是什么呢？它和视图查询类似，只不过内容查询是配合 `ng-content` 使用的，我们用它来获取不在组件模板里定义的元素。比如有一个列表组件 `MyListCpomponent`，它允许用户嵌入自定义内容，如果使用局部变量（以“#”号为前缀的属性）来定位该自定义内容上的元素，那会使得自定义内容和容器组件之间产生一定的耦合，要避免这种情况，则可通过 CSS 选择器来帮助定位元素。

假设用户使用 `MyListCpomponent` 编写的代码如下：

```

<my-list>
  <li *ngFor="let item of items"> {{item}} </li>
</my-list>

```

要定义 CSS 选择器（用于获取上面的 `` 元素），需要借助于 `@Directive` 装饰器。`@Directive` 装饰器提供了 `selector` 属性，它定义了 `li` 这个 CSS 选择器，再结合 `@ContentChildren` 来过滤并获得 `` 元素。示例代码如下：

```

// 定义 li 选择器，命名为 ListItem
@Directive({ selector: 'li' })
export class ListItem {}

// 组件代码，使用 ng-content 允许嵌入内容
@Component({
  selector: 'my-list',
  template: `
    <ul>
      <ng-content></ng-content>
    </ul>
  `,
})
export class MyListCpomponent implements AfterContentInit {

```

```

@ContentChildren(ListItem) items: QueryList<ListItem>;

ngAfterContentInit() {
  // 此时便能对 items 进行操作了
}
}

```

以上代码和以下使用 queries 的代码是等价的：

```

@Component({
  selector: 'my-list',
  template: `
    <ul>
      <ng-content></ng-content>
    </ul>
  `,
  queries: {
    items: new ContentChildren(ListItem)
  }
})
export class MyList implements AfterContentInit {
  items: QueryList<ListItem>;

  ngAfterContentInit() {
    // ...
  }
}

```

animations

Angular 提供了便捷的动画定义方法，使用起来就像自定义标签那样简单。要使用 animations 元数据定义，需要先从 @angular/animations 引入一些用于动画的函数。示例代码如下：

```

import {
  trigger,
  state,
  style,
  transition,
  animate
} from '@angular/animations';

```

现在通过简单的例子来说明 `animations` 元数据如何使用。我们创建一个 `Animation-ExampleComponent` 组件，该组件只有一个按钮，按钮有“on”和“off”两种状态，默认是“on”状态，当点击按钮时，会切换到“off”状态。注意到“on”状态时按钮文字是绿色的，并且比“off”状态时要大一些。

定义一个名称为 `buttonStatus` 的动画，它具有两种状态：“on”和“off”。当状态是“on”时，按钮显示为绿色，并且以 1.2 倍放大；当状态是“off”时，按钮显示为红色，并且恢复到 1 倍大小。同时我们定义了状态由“on”变成“off”，以及状态由“off”变成“on”的转换动画，这一切都可以使用 Angular 提供的方法来实现。示例代码如下：

```
// ...
animations: [
  trigger('buttonStatus', [
    state('on', style({
      color: '#0f2',
      transform: 'scale(1.2)'
    })),
    state('off', style({
      color: '#f00',
      transform: 'scale(1)'
    })),
    transition('off => on', animate('100ms ease-in')),
    transition('on => off', animate('100ms ease-out'))
  ])
]
// ...
```

这些语义化的代码可以帮助我们更好地理解 Angular 动画设置。这里定义了 `buttonStatus` 这个动画，接下来还需要在模板中使用它。Angular 提供了很便捷的语法，通过 `@triggerName`（例如以上代码定义的 `@buttonStatus`）的方式应用到元素中。示例代码如下：

```
// ...
template: `
  <div>
    <button @buttonStatus="status" (click)="toggleStatus()">
      {{status}}
    </button>
  </div>
`
// ...
```

通过 @buttonStatus 设置完成后, 点击按钮便能看到按钮状态变化的动画效果了! 完整的示例代码如下:

```
import {
  Component,
  trigger,
  state,
  style,
  transition,
  animate
} from '@angular/animations';

@Component({
  selector: 'animations-example',
  styles: [
    `button {
      width: 100px;
      height: 20px;
      background-color: #666;
      font-size: 15px;
    }`
  ],
  template: `
    <div>
      <button @buttonStatus="status" (click)="toggleStatus()">
        {{status}}
      </button>
    </div>
  `,
  animations: [
    trigger('buttonStatus', [
      state('on', style({
        color: '#0f2',
        transform: 'scale(1.2)'
      })),
      state('off', style({
        color: '#f00',
        transform: 'scale(1)'
      })),
      transition('off => on', animate('100ms ease-in')),
    ]),
  ],
})
```

```
        transition('on => off', animate('100ms ease-out'))
      ])
    })
  }
}

export class AnimationsExampleComponent {
  status: string = 'on';

  toggleStatus() {
    this.status = (this.status === 'on') ? 'off' : 'on';
  }
}
```

encapsulation

在本章的开头部分，提到了视图包装这个特性，它的主要目的是让组件的样式之间更加独立而互不影响，使得组件间的复用变得更简单。

读者可能在多人开发的项目中也遇到过样式冲突的情况，比如为元素 A 设置了 `.bg-style` 样式类，其作用是将背景设置为灰色；然而，另外一个开发者在其他地方也定义了同名（`.bg-style`）的样式类，其作用是将背景设置为红色，由于样式覆盖的效果，此时上面提到的元素 A 的背景并不会显示成原来的灰色，而是变成红色。在多人协作的项目开发中，难免会遇到一些样式命名冲突的情况，Angular 的视图包装特性正是用于解决这类问题的。下面用一个简单的例子来说明视图包装是如何做到隔离组件样式，并以此来规避命名冲突问题的。

视图包装可通过 `encapsulation` 这个元数据来设置，它允许设置以下三个可选值。

- `ViewEncapsulation.None`：无 Shadow DOM，并且也无样式包装。
- `ViewEncapsulation.Emulated`：无 Shadow DOM，但是通过 Angular 提供的样式包装机制来模拟组件的独立性，使得组件的样式不受外部影响，这是 Angular 的默认设置。
- `ViewEncapsulation.Native`：使用原生的 Shadow DOM 特性。

ViewEncapsulation.None

它表示 Angular 不使用 Shadow DOM，这时组件的样式会被写到 `document` 头部 `<header>` 标签里。

假设有一个 `HelloComponent` 组件，示例代码如下：

```
import { ViewEncapsulation } from '@angular/core';

@Component({
  selector: 'hello',
  template: `
    <div class="hello">
      <h1>Hello World</h1>
    </div>
  `,
  styles: [`
    .hello {
      background: green;
    }
  `],
  encapsulation: ViewEncapsulation.None
})
class Hello {
}
```

上面的代码使用了 `ViewEncapsulation.None` 选项值，Angular 会将组件模板渲染成：

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .hello {
        background: green;
      }
    </style>
  </head>
  <body>
    <hello>
      <div class="hello">
        <h1>Hello World</h1>
      </div>
    </hello>
  </body>
</html>
```

`ViewEncapsulation.None` 设置的结果是没有 Shadow DOM，并且所有的样式都会应用到整个 document。换句话说，组件的样式可以被覆盖。

ViewEncapsulation.Emulated

这是一个强大的功能，组件间的样式是相互独立、互不影响的，它是怎么做到的呢？还是以刚才的例子来说明，使用了这个设置后，Angular 会将组件模板渲染成：

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .hello[_ngcontent-1] {
        background: green;
      }
    </style>
  </head>
  <body>
    <hello _ngcontent-0 _ngghost-1>
      <div class="hello" _ngcontent-1>
        <h1>Hello World</h1>
      </div>
    </hello>
  </body>
</html>
```

虽然样式仍然应用到整个 document，但是 Angular 为 .hello 创建了一个 [_ngcontent-1] 选择器。组件的样式选择被 Angular 修改了，并且在组件的属性里也添加上了这个选择器。那 _ngcontent-0 和 _ngghost-1 又是干什么用的呢？为了实现局部的样式，Angular 需要保证某个组件的样式只会匹配到该组件，所以给组件添加了 _ngcontent-0 和 _ngghost-1 属性，其作用相当于命名空间。

ViewEncapsulation.Native

它表示使用原生的 Shadow DOM 特性。理解这个就非常简单了，Angular 会把组件以浏览器支持的 Shadow DOM 形式渲染，像刚才的例子在浏览器中能看到这样的结构：

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <hello>
      #shadow-root
      | <style>
      |   .hello {
```

```

|     background: green;
|   }
| </style>
| <div class="hello">
|   <h1>Hello World</h1>
| </div>
</hello>
</body>
</html>

```

6.7.3 深入理解 Zone.js

Zone.js 的设计理念来源于 Dart 语言，它描述的是 JavaScript 执行上下文，并且可以在异步任务之间持续传递。先抛开 Angular，Zone.js 到底可以解决什么样的问题？举一个简单的例子来说明，示例代码如下：

```

a();
setTimeout(b, 1000);
Promise.resolve().then(c);
d();

```

这是一段普通的 JavaScript 代码，很显然，a 和 d 函数先执行，然后是异步函数 c 和 b。假如我们需要统计上述脚本的总执行时间，如果仅在脚本的前后埋点，显然会错过异步函数 b 和 c。示例代码如下：

```

start(); // 开始计时
a();
setTimeout(b, 1000);
Promise.resolve().then(c);
d();
stop(); // 结束计时

```

上述埋点逻辑只能统计 a 和 d 函数的耗时，不符合预期。如果在 b 和 c 函数外围包装一层函数来统计，固然可以，但显得累赘，而且对业务逻辑代码有注入污染。但有了 Zone.js，这个问题就很好解决了。示例代码如下：

```

// 通过fork操作创建子 Zone，并添加计时函数
let childZone = Zone.current.fork({
  name: 'my-child-zone',
  onInvokeTask: function (delegate, current, target, task, applyThis, applyArgs) {
    start(); // 异步任务执行前
    delegate.invokeTask(target, task, applyThis, applyArgs);
  }
});

```



```
    stop(); // 异步任务执行后
  },
});

childZone.run(() => {
  start();
  a();
  setTimeout(b, 1000);
  Promise.resolve().then(c);
  d();
  stop();
});
```



Zone.current 是一个指向当前 Zone 实例的引用。Zone.js 初始化完成后，往 window 对象中添加了一个名为 Zone 的全局变量，同时会生成一个默认的 Zone 实例，即为 Root Zone。在默认情况下，Zone.current 指向 Root Zone。

首先通过 fork 操作生成额外配置的子 Zone，然后将脚本放置到子 Zone 下运行即可。可以看到，在这个额外配置里面定义了 onInvokeTask 钩子，这个钩子会在异步任务（如上例中的 setTimeout 和 Promise 回调函数）执行时被触发，并且可以在异步任务执行前后添加额外的逻辑，如上例添加的是一些计时函数。



除了 onInvokeTask，关于更多的 Zone 配置参数可以查阅 Zone.js 的 Github 代码库（/lib/zone.ts）。

这就是 Zone.js 跟踪异步任务执行上下文特性的强大之处，除可以帮助开发者分析性能外，还可以更方便地跟踪调试异步任务，或者进行测试等。

那么 Zone.js 为什么会有这样的能力？这里面到底有什么样的“黑魔法”？请继续阅读。

Zone.js 的原理

Zone 对象的 run() 方法是重要的入口方法之一，抽取 run() 方法的源码如下：

```
public run<T>(  
  callback: (...args: any[]) => T, applyThis: any = undefined, applyArgs: any[] =
```

```
    null,  
    source: string = null): T {  
// 1. 重新赋值_currentZoneFrame  
_currentZoneFrame = {parent: _currentZoneFrame, zone: this};  
try {  
    // 2. 执行回调函数  
    return this._zoneDelegate.invoke(this, callback, applyThis, applyArgs, source);  
} finally {  
    // 3. 还原_currentZoneFrame  
    _currentZoneFrame = _currentZoneFrame.parent;  
}  
}
```



`_currentZoneFrame` 是模块局部变量，在模块里的任何地方都可以引用，它记录的是当前在执行的脚本对应的 Zone 对象。初始化时 `_currentZoneFrame` 被赋值为 `{ parent: null, zone: [Root Zone Instance] }`。上述提到的 `Zone.current` 就是指向 `_currentZoneFrame.zone` 的值。

简单梳理一下，`run()` 方法大致进行了以下三步操作。

(1) 重新赋值 `_currentZoneFrame`，`parent` 指向老的 `_currentZoneFrame`，`zone` 指向 `this`，即当前的实例 Zone 对象。

(2) 执行相关的钩子函数及对应的回调函数。

(3) 将 `_currentZoneFrame` 还原为调用该回调函数之前的状态。

可以看到，`run()` 的关键一步是使用 `_currentZoneFrame` 保存了当前的 Zone 对象信息，Zone 对象保存的是异步任务的上下文信息，这些上下文信息可以在异步任务里持续传递。那么 `setTimeout()` 函数做了什么事情来持续传递当前的 Zone 对象呢？我们继续往下看。

Zone.js 在初始化的时候，还做了一件很重要的事情，即以猴子补丁的方式，在运行时替换浏览器的异步事件 API，如 `setTimeout`、`XHR`、`Promise` 等。所以当 Zone.js 初始化完成后，我们所使用的 `setTimeout` 等这些 API 都不是浏览器原来的 API 了，而是经过 Zone.js 包装之后的新 API。很显然，Zone 对象的传递是在包装函数中完成的，包装函数通过将回调函数与当前的 Zone 实例 (`Zone.current`) 进行绑定生成 `ZoneTask` 对象 (`ZoneTask` 是 Zone.js 内部记录异步任务上下文数据的一个重要对象)，这样二者便形

成了关联关系。当回调函数被触发的时候便运行 ZoneTask，ZoneTask 运行时的关键流程跟上述提到的 run() 方法类似，首先将当前的 Zone 对象指向 ZoneTask 里绑定的 Zone 对象，然后执行相关的钩子函数及回调函数，最后把当前的 Zone 对象还原为运行前的 Zone 对象。

基于上述说明，为示例代码加一些调试信息，修改代码如下：

```
console.log(`1: ${Zone.current.name}`);
childZone.run(() => {
  console.log(`2: ${Zone.current.name}`);
  a();
  setTimeout(function() {
    console.log(`3: ${Zone.current.name}`);
    b();
  }, 1000);
  Promise.resolve().then(function() {
    console.log(`4: ${Zone.current.name}`);
    c();
  });
  d();
  console.log(`5: ${Zone.current.name}`);
});
console.log(`6: ${Zone.current.name}`);
```

输出结果为：

```
1: <root>
2: my-child-zone
a()
d()
5: my-child-zone
6: <root>
4: my-child-zone
c()
3: my-child-zone
b()
```

Zone 的继承

虽然不同的 Zone 具有一定的隔离性，但是若两个 Zone 是父子关系，那么它们之间便存在着继承关系，并遵循 JavaScript 的原型链数据传递规则。Zone 的每个函数在被执

行时都会创建自己的执行环境，当代码在某一执行环境中运行时，会创建由变量对象构成的一个作用域链，这确保了执行环境对所有变量或者函数的有序访问。下面我们通过例子来理解 Zone 的继承，示例代码如下：

```
let zoneParent = Zone.current;
let zoneA = zoneParent.fork({ name: 'zoneA', properties: { x: 1, y: 1 } });
let zoneB = zoneA.fork({ name: 'zoneB', properties: { x: 2 } });

console.log('zoneA 属性 x 的值是: ', zoneA.get('x')); // 1
console.log('zoneA 属性 y 的值是: ', zoneA.get('y')); // 1
console.log('zoneB 属性 x 的值是: ', zoneB.get('x')); // 2

// 当当前 Zone 获取不到某属性时，将向父 Zone 查询该属性
console.log('zoneB 属性 y 的值是: ', zoneB.get('y')); // 1
```

6.7.4 不依赖 Zone.js 的 Angular

Zone.js 使得开发者在开发时无须关注数据何时发生了变化，但 Zone.js 也不是万能的，它有一些存在已久的问题，一是备受诟病的稳定性问题；二是 Zone.js 补丁（polyfill）会增加应用包体的大小；三是性能还有待提升，在一些对性能要求比较高的场景中容易导致性能瓶颈。所以在 5.0 版本之后，Angular 开始支持不依赖 Zone.js 的用法，将变化监测的主动权完全开放给开发者。

要移除 Zone.js，只需在应用启动时告诉 Angular 即可。示例代码如下：

```
platformBrowserDynamic()
  .bootstrapModule(AppModule, {ngZone: 'noop'}); // 增加 {ngZone: 'noop'} 配置项
```

增加了 {ngZone: 'noop'} 配置项后，即可移除 Zone.js 的相关 polyfill 而应用不会报错。但没有了 Zone.js 的帮忙，此时应用的界面就不会自动更新了，必须要手动触发更新，可以手动调用 ChangeDetectorRef.detectChanges() 来实现。示例代码如下：

```
class AppComponent {
  name = 'with zones';
  constructor(private cd: ChangeDetectorRef) {
    setTimeout(() => {
      // 修改应用数据操作
      this.name = 'without zones';
      // 手动执行变化监测
      cd.detectChanges();
    }, 1000);
  }
}
```

```
}  
}
```

需要注意, `ChangeDetectorRef.detectChanges()` 只能更新本组件及其子组件树, 如果对状态的修改涉及其他组件树分支, 则需要使用 `ApplicationRef.tick()` 来触发整棵组件树的变化监测执行。示例代码如下:

```
constructor(private appRef: ApplicationRef) {  
  setTimeout(() => {  
    // 修改应用数据操作  
    this.name = 'without zones';  
    // 手动执行变化监测  
    appRef.tick();  
  }, 1000);  
}
```



最好不要频繁地触发执行 `ApplicationRef.tick()`, 在实际应用中需要再做进一步封装, 例如将 `tick()` 操作放到 `requestAnimationFrame` 的回调函数里执行, 或者进行其他更智能化的批量操作。

6.8 小结

本章首先介绍了组件的由来, 接着重点讲述了 Angular 组件的技术细节, 以及组件和与其相关的其他组成部分的关联关系。文中详细介绍了组件的基础知识, 包括常用的元数据, 如 `selector`、`template / templateUrl`、`styles / styleUrls` 等, 以及组件类与模板的数据交互和组件间的数据交互, 并介绍了如何通过使用 `<ng-content>` 实现组件内容嵌入功能。接下来对组件的生命周期及组件变化监测机制也做了较为详细的讲解, 列举了各个生命周期钩子的执行时机, 分别讲述了异步事件的获取 (NgZone, 源于 Zones)、组件数据的变化监测, 以及响应变化监测的执行细节等内容。作为扩展阅读, 也罗列了组件的大部分元数据, 并在最后对 Zones 机制做了进一步的讲解。

组件是 Angular 应用的最重要的组成部分, 通过对本章内容的学习, 相信读者对组件也有了一定的了解。Angular 的很多概念都能在组件中体现, 本章在某些地方引出了模块、模板、指令、服务、依赖注入、路由等概念, 这些概念都会在后续章节中详细阐述。接下来我们将介绍与组件最密切的模板部分。



模板

在第 6 章中我们学习了组件相关知识，本章将学习 Angular 模板相关内容。

模板是一种自定义的标准化页面，通过模板和模板中的数据结合，可以生成各种各样的网页。在 Angular 中，模板的默认语言是 HTML，几乎所有的 HTML 语法在模板中都是适用的，但 `<script>` 标签是被禁止的，主要是为了防止 JavaScript 脚本注入攻击（即 XSS）。同时一些 HTML 元素在模板中并不起什么作用，比如 `<html>`、`<body>`、`<base>` 等。除此之外，在 Angular 中可以通过组件和指令对模板的 HTML 元素进行扩展，这些扩展将以新的元素或属性的形式出现。

在这一章中，首先学习在模板中如何通过数据绑定实现模板和组件之间的数据流动、数据绑定的几种方式、什么是输入和输出属性；然后介绍一些典型内置指令的使用方法；接下来讲解在 Angular 中如何实现与表单相关的校验、提交、数据追踪及绑定、表单状态的样式修改等；最后学习管道及管道的应用场景等知识。

7.1 模板语法概览

在深入学习模板之前，先来初步认识一下 Angular 中模板的语法，如表 7-1 所示。

表 7-1 模板语法概览

示例	名称	说明	语法	详解章节
<code><p>{{ detail.telNum }}</p></code>	插值	绑定属性变量的值到模板中	<code>{{ 模板表达式 }}</code>	7.2.2 节
<code><div [title]="name">hello world</div></code>	DOM 元素属性绑定	将模板表达式 <code>name</code> 的值绑定到元素 <code><div></code> 的属性 <code>title</code> 上	<code>[DOM 元素属性]= "模板表达式" 或者 bind-DOM元素属性="模板表达式"</code>	7.2.4 节
<code><td [attr.colspan]="{{ 1 + 2 }}">合并单元格</td></code>	HTML 标签特性绑定	将模板表达式的返回值绑定到元素 <code><td></code> 的标签特性 <code>colspan</code> 上	<code>[attr.HTML 标签特性]="模板表达式"</code>	7.2.4 节
<code><div [class.isblue]="isBlue()">单击后这里将变为14号蓝色字</div></code>	CSS 类绑定	当 <code>isBlue()</code> 函数值为 <code>true</code> 时, 为 <code>div</code> 添加类名为 <code>isblue</code> 的样式	<code>[class.css 类名] ="模板表达式"</code>	7.2.4 节
<code><button [style.color]="isRed?'red':'green'">红色</button></code>	Style 样式绑定	当表达式 <code>isRed</code> 的值为 <code>true</code> 时, 设置 <code>button</code> 的文字颜色为红色, 否则为绿色	<code>[style.css 样式属性名]="模板表达式"</code>	7.2.4 节
<code>编辑</code>	事件绑定	点击元素时会触发 <code>click</code> 事件, 在需要时也可以传递 <code>\$event</code> 对象, 如 <code>(click)="editContact(\$event)"</code>	<code>(事件)="模板语句" 或者 on-事件="模板语句"</code>	7.2.5 节
<code><div [(title)]= "name"></div></code>	双向绑定	组件和模板之间双向数据绑定, 等价于 <code><div [title]="name" (titleChange)="name=\$event"></div></code>	<code>[(绑定目标)]= "模板表达式" 或者 bindon-绑定目标="模板表达式"</code>	7.2.6 节
<code><input type="text" #name><p>{{ name.value }}</p></code>	模板局部变量	创建目标元素或组件的引用变量, 如此处的 <code>name</code> , 该变量在当前模板内均可使用	<code>#变量名 或者 ref-变量名</code>	7.4.2 节
<code><p>张三的生日是 {{ birthday date }}</p></code>	管道操作符	原始数据 <code>birthday</code> 经管道转换后输出期望数据并显示在模板中	<code>输入数据 管道名 :管道参数</code>	7.5.1 节

续表

示例	名称	说明	语法	详解章节
<code><p>{{ detail?.telNum }}</p></code>	模板表达式操作符	模板表达式操作符表明 <code>detail</code> ？. <code>.telNum</code> 属性不是必须存在的，如果它的值是 <code>undefined</code> ，那么后面的表达式将会被忽略，不会引发异常		7.6 节
<code><p *myUnless="boolValue">myUnless is false now.</p></code>	星号前缀	使用星号前缀可以简化对结构指令的使用，Angular 会将带有星号的指令引用替换成带有 <code><ng-template></code> 标签的代码，等价于 <code><ng-template [myUnless]="boolValue"><p>myUnless is false now.</p></ng-template></code>	*指令	8.3.2 节

上述模板语法在后续章节中都会进行详细讲述，表 7-1 中第 5 列即是对应的章节。

7.2 数据绑定

7.2.1 概述

数据绑定为应用程序提供了一种简单、一致的机制来管理与协调数据的显示，以及数据值的变化。这种机制可以从 HTML 里面取值和赋值，使得数据的读写变得更加简单、快捷。Angular 提供了多种数据绑定方式，可以根据数据流动的方向分为三种，详见表 7-2。

表 7-2 数据绑定方式

数据流向	示例	绑定类型
单向：从数据源到视图目标（属性绑定）	<code><p>{{ detail.telNum }}</p></code> <code><div [title]="name">hello world</div></code> <code><div [style.color]="color">hello world</div></code>	插值 DOM 元素属性绑定，HTML 标签特性绑定
单向：从视图目标到数据源（事件绑定）	<code>(click)="editContact()"</code> <code>on-click="editContact()"</code>	事件绑定

续表

数据流向	示例	绑定类型
双向	<code><div [(title)]= "name"></div> <div bindon-title = "name"></code>	双向绑定

通过图 7-1 可以更直观地看出数据绑定的数据流向（箭头表示数据的流向）。

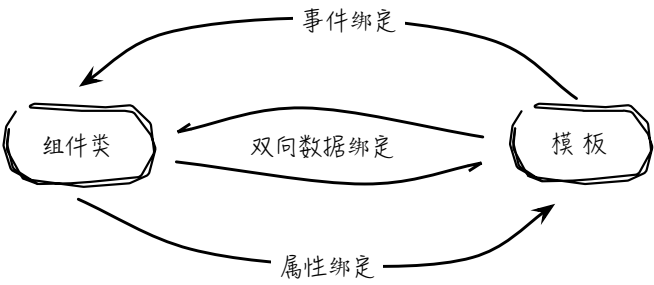


图 7-1 数据绑定的数据流向

在上述绑定类型中，除插值外，在“=”的左侧都会有一个目标名称，它可以被 []、() 包裹，或者加上一个前缀（bind-、on-、bindon-），这被称为绑定目标。而“=”右侧或者插值符号“{{}}”中的部分则被称为绑定源。

在学习数据绑定前先来了解一对非常重要的概念，即 DOM 对象属性（Property）和 HTML 标签特性（Attribute）。在英语中 Property 和 Attribute 都可以译为“属性”，名字虽然相同，但在模板中意义却大有不同。理解这两个属性的不同，是理解 Angular 数据绑定的关键。这里我们将 Property 称为 DOM 对象属性，而把 Attribute 称为 HTML 标签特性，具体的定义如表 7-3 所示。

表 7-3 DOM 对象属性和 HTML 标签特性的定义

定义方式	说明
DOM 对象属性（Property）	以 DOM 元素作为对象，其附加的内容是在文档对象模型里定义的，如 childNodes、firstChild 等
HTML 标签特性（Attribute）	DOM 节点自带的属性，是在 HTML 里定义的，即只要是在 HTML 标签中出现的属性（HTML 代码）都是 Attribute，例如 HTML 中常用的 colspan、align 等

此外，两者的区别和联系还体现在以下两个方面。

- 在大多数情况下，DOM 对象属性与 HTML 标签特性并不是一一对应的，但有少量属性既是 DOM 对象属性又是 HTML 标签特性，如 id、title、class（CSS 类）等。
- 通常 HTML 标签特性代表着初始值，初始化后就不再发生改变；而 DOM 对象属性代表着当前值，默认为初始值，但它会随着属性值的变化而变化。

数据绑定是借助于元素与指令的 DOM 对象属性和事件来运作的，而不是 HTML 标签特性。在 Angular 中，HTML 标签特性的唯一作用就是用来进行元素和指令状态的初始化。

7.2.2 插值

数据绑定最常见的形式就是插值（Interpolation），默认使用的是双大括号“{{ }}”的语法。使用插值可以在 HTML 元素标签和属性值内将变量输出。示例代码如下：

```
<li>
  <p> 手机号码: </p>
  <p>{{ detail.telNum }}</p>
</li>
```

双大括号中间的值通常是一个组件属性的变量名，Angular 使用相应组件属性的值来替换这个变量。在以上例子中，Angular 会对 detail.telNum 求值并替换后，最终在页面中将结果展示出来。双大括号里面还可以是一个合法的模板表达式，Angular 会首先进行求值，然后转换成字符串输出到页面上。示例代码如下：

```
<!-- 512+512 = 1024 -->
<p>512 + 512 = {{512 + 512}}</p>
```

表达式甚至可以调用宿主组件的函数，如用 getName() 方法获取某个联系人的姓名。示例代码如下：

```
<p>{{detail.getName()}}</p>
```

7.2.3 模板表达式

模板表达式类似于 JavaScript 语言的表达式，绝大部分的 JavaScript 表达式都是合法的模板表达式。模板表达式应用在插值语法的双大括号中和属性绑定“=”右侧的引号中。Angular 会执行这个表达式并将值分配给一个绑定目标的属性，这个绑定目标可能

是一个 HTML 元素、组件或者指令。但不可以使用以下可能引发副作用的 JavaScript 表达式：

- 带有 new 运算符的表达式。
- 赋值表达式（=、+=、-=等）。
- 带有“;”或者“,”的链式表达式。
- 带有自增和自减操作（++ 和 --）的表达式。

其他与 JavaScript 语法不同且值得注意的特性包括：

- 不支持位运算符（| 和 &）。
- 部分模板表达式操作符被赋予了新的含义，如管道操作符（|）和安全导航操作符（?.）等。

模板表达式上下文

模板表达式的上下文通常就是它所在组件的实例，也可以包括组件之外的对象，如模板局部变量。在通讯录例子的联系人详情页面（detail.component.html）中，模板表达式 {{detail.telNum}} 的上下文就是它所在组件的实例。

需要特别注意的是，模板表达式不能引用任何全局命名空间中的成员，如 window 和 document，也不能调用 console.log() 或者 Math.random() 等方法。

模板表达式的书写原则

在书写模板表达式时要注意以下几条原则。

- 避免视图变化的副作用。一个模板表达式只能改变目标属性的值，不应改变应用的任何状态，Angular 的“单向数据流”模式正是基于这条原则而来的。在单独的渲染过程中，视图应是可预测到的，不必担心在读取组件值时会不小心改变其他的一些展示值。
- 能够高效地执行。Angular 执行表达式的频率远超我们的想象，触发任何一次的键盘或者鼠标事件，这些表达式都可能会被执行。当计算的成本比较大时，可以考虑缓存那些从其他值计算得出的值。
- 使用简单的语句。避免编写一些比较复杂的模板表达式。
- 幂等性优先。表达式应遵循幂等性优先原则。幂等的表达式总是会返回完全一致的东西，这样就没有副作用了，并能提升 Angular 变化监测的性能，但表达式的返回值还是会随着它所依赖的值变化而变化的。

7.2.4 属性绑定

属性绑定是一种单向的数据绑定，数据从组件类流向模板。当要把一个视图元素的属性设置为模板表达式时，就需要用到模板的属性绑定。

属性绑定不能用来从目标元素获取值，或者调用目标元素的方法。也就是说，目标元素的值只能被设置，不能被读取。但是我们可以使用 `@ViewChild` 和 `@ContentChild` 来读取目标元素的属性或调用它的方法，具体示例可参阅第 6 章。

DOM 元素属性绑定

最常用的属性绑定是把元素的属性绑定到组件的属性上。在下面的例子中，`div` 元素的 `title` 属性会被绑定到组件的 `titleText` 属性上。

```
<div [title]="titleText">hello world</div>
```

此时 DOM 对象属性（`title`）就是绑定目标，也可以选择 `bind-` 前缀的形式来实现属性绑定，这种绑定又称为标准形式。示例代码如下：

```
<div bind-title="titleText">hello world</div>
```

下面例子是用来设置 `Angular` 指令的属性的。

```
<div [ngStyle]="styles"> [ngStyle] 绑定到 styles 属性 </div>
```

此外，还可以使用属性绑定设置自定义组件的输入属性（这是父子组件间通信的重要途径）。示例代码如下：

```
<user-detail [user]="currentUser"></user-detail>
```

中括号

在属性绑定中，“=” 左侧中括号的作用是让 `Angular` 执行 “=” 右侧的模板表达式，并将结果赋值给该目标属性。如果没有中括号，`Angular` 就会把 “=” 右侧的模板表达式当作一个字符串常量，而不会计算该字符串。所以，如果赋值给目标属性的值是一个固定的字符串，那么推荐省略中括号。

在标准的 HTML 中常用这种方式来初始化 HTML 标签特性（`Attribute`）的值，在 `Angular` 中也可以用这种方式来初始化指令和组件属性的值。下面例子把 `detail` 属性初始化为一个固定的字符串，而不是模板表达式。`detail` 属性的值在初始化之后将不再改变，而通过属性绑定的 `user` 将会随着模板表达式的值变化而变化。示例代码如下：

```
<user-detail detail="我是字符串常量" [user]="currentUser"></user-detail>
```

HTML 标签特性绑定

Angular 推荐使用 DOM 元素属性绑定，但当元素没有对应的属性可绑定的时候，则可以使用 HTML 标签特性绑定来设置值。例如 `<table>` 中的 `colspan` 或 `rowspan` 等 HTML 标签特性，是纯粹的 HTML 标签特性，并没有相对应的 DOM 元素属性可供绑定，如果直接用模板表达式赋值，如下例所示：

```
<table border=1>
  <!--...-->
  <tr><td colspan="{{ 1 + 2 }}"> 合并单元格 </td></tr>
</table>
```

这将会出现一个模板解析的错误，因为 `colspan` 在 `<td>` 元素中并不是 DOM 元素属性，而是 HTML 标签特性。插值和属性绑定只能设置 DOM 元素属性，不能设置 HTML 标签特性。

HTML 标签特性绑定在语法上类似于属性绑定，但中括号中的部分不是一个元素的属性名，而是由 `attr.` 前缀和 HTML 标签特性名称组成的形式，然后通过一个模板表达式来设置 HTML 标签特性的值的。上面的例子可以进行如下改造：

```
<table border=1>
  <tr><td [attr.colspan]="{{ 1 + 2 }}"> 合并单元格 </td></tr>
</table>
```

CSS 类绑定

正如上文所提到的，CSS 类既属于 HTML 标签特性，又属于 DOM 对象属性，所以可以使用以上两种方式来完成属性绑定。示例代码如下：

```
<!-- 标准 HTML 样式类设置 -->
<div class="font14">14 号红色字 </div>

<!-- 通过绑定重设或覆盖样式类 -->
<div class="red font14" [class]="changeGreen">14 号绿色字 </div>
```



在第二个例子中，当使用 DOM 对象属性绑定给 `[class]` 绑定值时，`changeGreen` 对象会重写这个 `div` 元素的全部 `class`。

另外，Angular 也给 CSS 类属性绑定提供了特有的绑定方式，即使用类似于 `[class.class-name]` 的语法形式来完成属性绑定——当被赋值为 `true` 时，将 `class-name` 这个类

添加到该绑定的标签上，否则将移除这个类。示例代码如下：

```
<!-- 通过一个属性值来添加或移除 CSS 类 -->
<div [class.color-blue]="isBlue()">
    若 isBlue() 返回 true，这里的字体将变为蓝色的
</div>

<div class="footer" [class.footer]="showFooter">
    若 showFooter 为 false，则 footer 这个 CSS 类将被移除
</div>
```

Style 样式绑定

HTML 标签内联样式可以通过 Style 样式绑定的方式来设置。样式绑定在语法上采用形如 [style.style-property] 的写法。如设置按钮的背景色为蓝色，示例代码如下：

```
<button [style.background-color]="canClick ? 'blue' : 'grey'" >
    若 canClick 为 true，则按钮的背景色为蓝色
</button>
```

在设置内联样式时也可以带上样式单位，如 % 和 px 等。示例代码如下：

```
<!-- 带有单位的样式绑定 -->
<button [style.font-size.px]="isLarge ? 18 : 13" >
    若 isLarge 为 true，则按钮的字体将变成 18px
</button>
<div [style.width.%]="!isHalf ? 100 : 50" >
    若 isHalf 为 true，则该元素的宽度变成 50%
</div>
```



样式属性可以采用“烤肉串”命名法（如 font-size），也可采用“驼峰式”命名法（如 fontSize）。

属性绑定与插值的关系

属性绑定和插值都能实现数据绑定，在下面的例子中，属性绑定和插值两者实现的效果是一样的。

```
<div>hello world, <i>{{ userName }}</i></div>
<div>hello world, <i [innerHTML]="userName"></i></div>
```

属性绑定和插值在本质上没有区别，在渲染视图之前，Angular 会将插值表达式转换成属性绑定的形式，它只是属性绑定的一种语法糖。在大部分情况下，基于使用的便捷性与可读性的考虑，我们更推荐使用插值表达式。不论哪种方式，Angular 都会对不安全的 HTML 有所防备。在渲染显示它们之前，Angular 会先对内容进行“安全处理”，不会允许带有 `<script>` 标签的 HTML 展示到浏览器中。如下面的写法是会被过滤的：

```
<p>{{ userName }}</p>
```

```
<p>上面 userName 变量的内容是 `模板 <script>alert("我带有攻击性")</script> 语法`</p>
```

7.2.5 事件绑定

事件绑定也是一种单向数据绑定形式，数据从模板流向组件类。在事件绑定中，Angular 通过监听用户操作事件，比如键盘事件、鼠标事件、触屏事件等来执行其对应绑定的方法。事件绑定的语法是由“=”左侧小括号内的目标事件和“=”右侧引号中的模板语句组成的。示例代码如下：

```
<a class="edit" (click)="editContact()"> 编辑 </a>
```

在上面例子中，用事件绑定来监听按钮的点击事件，只要触发点击事件，就会调用组件的 `editContact()` 方法。

模板语句

上文提到的事件绑定“=”右侧的部分是模板语句，它是用来响应由绑定目标（如 HTML 元素或指令）所触发的事件的。模板语句可以帮助我们实现接收用户的输入来更新应用的状态。模板语句和模板表达式一样，与 JavaScript 表达式类似，但两者的解析器是不同的，模板语句除支持“=”赋值操作外，也支持用分号或者逗号串联起多条语句。当然，有一些 JavaScript 表达式在模板语句中是不被支持的，例如：

- 赋值操作，如 `+=` 和 `-=`。
- 自增和自减操作符（`++` 和 `--`）。
- `new` 操作符。
- 位运算符“`|`”和“`&`”。
- 模板表达式运算符。

模板语句和模板表达式一样，只能访问其上下文环境中的成员，模板语句的上下文环境就是绑定事件对应组件的实例。模板语句的上下文也可以包含组件之外的对象，如模板局部变量和事件绑定语句中的 `$event`。

目标事件

在小括号 “()” 中的事件名表示目标事件，如下例中的 click 事件是该事件绑定的目标事件。示例代码如下：

```
<a class="edit" (click)="editContact()"> 编辑 </a>
```

除了小括号，也可以使用带 on- 前缀的形式来标记目标事件。示例代码如下：

```
<a class="edit" on-click="editContact()"> 编辑 </a>
```

目标事件可以是常见的元素事件（如 click），也可以是自定义指令的事件。示例代码如下：

```
<a class="edit" (myClick)="editContact=$event"> 编辑 </a>
```

Angular 在解析目标事件时，会优先判断是否匹配已知指令的事件，如果事件名既不是某个已知指令的事件，也不是元素事件，Angular 就会抛出一个“未知指令”的错误。

\$event 事件对象

在 JavaScript 中，当用户点击某个元素时，就会触发元素绑定的事件，接着就会执行模板语句。这里可以通过 \$event 事件对象来获取该事件的相关信息，例如获取触发此事件的元素，以及事件发生位置的坐标等。目标事件的类型决定了事件对象的形态，目标事件可以是 DOM 元素事件，也可以是自定义事件。若目标事件是原生的 DOM 元素事件，则 \$event 将是一个包含 target 和 target.value 属性的 DOM 事件对象。示例代码如下：

```
<input [value]="currentUser.firstName" (input)="currentUser.firstName=$event.target.value" >
```

在上面例子中，当用户更改输入框中的文本时，input 事件会被触发，对应的模板语句就会被执行。此时，该模板语句的上下文中包含一个 \$event 对象，通过 \$event.target.value 可以将通过输入框输入的值绑定并修改 firstName 属性的值。

自定义事件

在 Angular 中，组件要触发自定义事件可以借助于 EventEmitter。在组件中可以创建一个 EventEmitter 实例对象，并将其以输出属性的形式暴露出来。父组件通过绑定这个输出属性来自定义一个事件，在组件中调用 EventEmitter.emit(payload) 来触发这个

自定义事件，其中 `payload` 可以传入任何值，父组件绑定的事件可以通过 `$event` 对象来访问 `payload` 的数据。

在通讯录例子的联系人列表页面中，通过点击对应联系人的区域，就可以看到联系人的详细信息。这里采用自定义事件来实现这种效果，示例代码如下：

```
// item.component.ts
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'list-item',
  templateUrl: 'app/list/item.component.html',
  styleUrls: ['app/list/item.component.css']
})
export class ListItemComponent {
  @Input() contact:any = {};
  @Output() routerNavigate = new EventEmitter<number>();

  goDetail(num: number) {
    this.routerNavigate.emit(num);
  }
}

<!-- item.component.html -->
<a (click)="goDetail(contact.id)">
  <!--...-->
</a>
```

组件 `ListItemComponent` 定义了一个 `EventEmitter` 的实例 `routerNavigate`。当点击 `<a>` 标签时，组件就会调用 `goDetail()` 方法，在这个方法里再执行 `EventEmitter.emit()` 方法，传递一个数字，并跳转到对应的联系人详情页面。另外，作为宿主的父组件绑定了 `ListItemComponent` 的 `routerNavigate` 事件。示例代码如下：

```
// list.component.html
<ul class="list">
  <li *ngFor="let contact of contacts">
    <list-item [contact]="contact" (routerNavigate)="routerNavigate($event)">
      </list-item>
    </li>
  </ul>
```

当 `routerNavigate` 事件被触发时，Angular 就会调用父组件的 `routerNavigate()` 方法，在 `$event` 中传入对应联系人的 `id`。执行 `routerNavigate()` 方法使得页面内容发生更新，进而显示联系人的详细信息。

7.2.6 双向数据绑定

上面提到的属性绑定和事件绑定都是单向数据绑定，而在实际开发中，有时也会碰到需要双向数据绑定的场景，例如在表单展示的功能中，一般需要将组件的数据显示到表单上，也需要将用户修改的数据更新到组件中。想实现这种双向绑定的效果，可以利用属性绑定和事件绑定结合的形式来处理。示例代码如下：

```
<input [value]="currentUser.firstName" (input)="currentUser.firstName=$event.target.value" >
```

在上面代码中，属性绑定实现了数据从组件类流向模板，事件绑定则实现了数据从模板流向组件类，将两者结合起来，就实现了双向绑定的效果。Angular 提供了 `NgModel` 指令可以更方便地进行双向绑定。示例代码如下：

```
<input  
  [ngModel]="currentUser.phoneNumber"  
  (ngModelChange)="currentUser.phoneNumber=$event">
```

`NgModel` 指令通过 `ngModel` 输入属性和 `ngModelChange` 输出属性隐藏了一些烦琐的细节。这样看起来虽然好一点，但还是不够，对同一个数据源进行两次数据绑定还是不够干脆。在 Angular 中可以使用一种更简洁的语法 “[()]” 来实现双向数据绑定，示例代码如下：

```
<input [(ngModel)]="currentUser.phoneNumber">
```

在上面代码中，“[]” 实现了数据流从组件类到模板，“()” 实现了数据流从模板到组件类，两者结合的 “[()]” 就可以很简单地实现双向绑定了。这种语法可以用从大到小的写法顺序来帮助记忆，即先写中括号，再写小括号。另外，它也可以采用前缀形式进行书写，示例代码如下：

```
<input bindon-ngModel="currentUser.phoneNumber">
```

“[()]” 这种语法只能设置一个数据绑定属性，若想完成更多不同的任务，就得采用它的展开形式来实现。例如想实现一个给联系人手机号码加上区号的功能，示例代码如下：

```
<input  
  [ngModel]="currentUser.phoneNumber"  
  (ngModelChange)="addCodeForPhoneNumber($event)">
```

更多的关于双向绑定的原理，可以在本章的“扩展阅读”部分进行更深入的了解。

7.2.7 输入和输出属性

正如上文所提到的，在绑定声明“=”右侧的部分，称之为数据绑定的源；而“=”左侧的部分，称之为数据绑定的目标，如下面的例子：

```
<!-- list.component.html -->
<list-item [contact]="contact" (routerNavigate)="routerNavigate($event)">
</list-item>
```

在上面代码中，位于绑定声明“=”左侧的 `contact` 和 `routerNavigate`，它们都是数据绑定的目标。其中，`contact` 是属性绑定的目标，`routerNavigate` 是事件绑定的目标。数据通过模板表达式流向目标属性 `contact`，那么 `contact` 在组件 `ListComponent` 中是一个输入属性。同样的，在事件绑定中，数据流向 `routerNavigate` 绑定源，传递给接收者，在组件 `ListComponent` 中 `routerNavigate` 是一个输出属性。

声明输入和输出属性

绑定目标必须被明确地标记为输入或输出属性。在“组件”章节中我们已经学习了输入、输出属性的两种声明方式，即 `@Input` 和组件元数据 `inputs`、`@Output` 和组件元数据 `outputs`，这里不再赘述，但切记不要在同一组件中同时采用装饰器（`@Input` 或 `@Output`）和元数据（`inputs` 或 `outputs`）这两种声明方式。

输入和输出属性别名

有时输入、输出属性名的语义不是很明确，可能描述不清这个属性的作用或者功能，因此给输入、输出属性定义一个有语义的别名是很有必要的。定义输入、输出属性别名有以下两种方式。

方式一：通过 `@Input` 或 `@Output` 装饰器为属性指定别名，语法形如“`@Output(别名)` 事件属性名 = ...”。比如给一个自定义事件定义一个别名 `goto`，示例代码如下：

```
@Output('goto') clicks = new EventEmitter<number>();
```

方式二：采用组件（指令）元数据的 `inputs` 或 `outputs` 数组可以为属性指定别名，语法形如“`outputs: ['组件属性名: 别名']`”。示例代码如下：

```
@Component({
  outputs: ['clicks:goto']
})
```

在组件中给输入、输出属性定义别名后，就可以直接在外部组件的数据绑定中使用这个别名了。

7.3 内置指令

在 Angular 中，指令作用在特定的 DOM 元素上，可以扩展这个元素的功能，为元素增加新的行为。Angular 框架本身自带一些指令，如 `NgClass`、`NgStyle`、`NgIf`、`NgSwitch`、`NgFor` 等，它们也被称为 Angular 内置指令。接下来将重点学习模板中常见的内置指令。

7.3.1 NgClass

正如上文所提到的，在属性绑定中，CSS 类绑定的方式能够为标签元素添加和移除单个类。在实际开发中，通过动态添加或移除 CSS 类的方式，可以控制元素的展示。在 Angular 中，通过 `NgClass` 指令可以同时添加或移除多个类。`NgClass` 绑定一个形如“CSS 类名: value”的对象，其中 value 是一个布尔类型的数据值，当 value 为 true 时，添加对应的类名到模板元素中；反之，则移除。

例如，在组件中设置一个管理类状态的对象，用来控制在模板元素中是否添加 `red`、`font14` 和 `title` 类。示例代码如下：

```
// ...
setClasses() {
  let classes = {
    red: this.red, // true
    font14: !this.font14, // false
    title: this.isTitle // true
  };
  return classes;
}
// ...
```

接下来，通过添加一个 `ngClass` 属性绑定，调用组件的 `setClasses()` 方法来设置该元素的类样式。示例代码如下：

```
<div [ngClass]="setClasses()"> 红色标题文字 </div>
```

7.3.2 NgStyle

正如上文所提到的，在属性绑定中，Style 样式绑定的方式能够给模板元素设置单一的样式。而采用 `NgStyle` 指令可以为模板元素设置多个内联样式，与 `NgClass` 类似，

NgStyle 绑定一个形如“CSS 属性名: value”的对象，其中 value 为具体的 CSS 样式。

例如，在组件中设置一些内联的 CSS 样式，用来在模板元素中设置 color、font-size 和 font-weight。示例代码如下：

```
// ...
setStyles() {
  let styles = {
    'color': this.red ? 'red' : 'blue', // red
    'font-size': !this.font14 ? '14px' : '16px', // 16px
    'font-weight': this.isSpecial ? 'bold' : 'normal' // bold
  };
  return styles;
}
// ...
```

接下来，通过使用 NgStyle 属性绑定的方式，调用 setStyles() 方法来设置该元素的内联样式。示例代码如下：

```
<div [ngStyle]="setStyles()"> 红色 16px 的加粗文字</div>
```

7.3.3 NgIf

通过 NgIf 指令绑定一个布尔类型的表达式，当表达式返回 true 时，可以在 DOM 树的节点上添加一个元素及其子元素；反之，将被移除。示例代码如下：

```
<h3 *ngIf="collect.length === 0" class="no-collection"> 未收藏 </h3>
```



*ngIf 是一种语法糖的写法，下文中的 NgSwitchCase、NgSwitchDefault 和 Ngfor 与此类似。欲了解更多详情请参阅 8.3.2 节。

NgIf 与类绑定、样式绑定的方式有什么区别呢？通过类绑定、样式绑定的方式也可以设置模板元素的显示与隐藏，如通过 class.hidden 属性绑定的方式可以控制是否显示该模板元素。示例代码如下：

```
<h3 [class.hidden]="collect.length === 0" class="no-collection"> 未收藏 </h3>
```

类绑定、样式绑定也具备显示或隐藏某个元素及其子元素的效果，但是与 NgIf 不同的是，它们仅仅设置了元素是否显示，而该元素还保留在 DOM 树的节点上，类似于加上了“display: none”的样式效果；而 NgIf 当表达式返回值为 false 时，元素是会从 DOM 树上移除的。

7.3.4 NgSwitch

NgSwitch 指令需要结合 NgSwitchCase 和 NgSwitchDefault 指令来使用, 根据 NgSwitch 绑定的模板表达式的返回值来决定添加哪个模板元素到 DOM 树的节点上, 并移除其他备选模板元素。这三个相互合作的指令分别表示:

- ngSwitch, 绑定到一个返回控制条件的值的表达式。
- ngSwitchCase, 绑定到一个返回匹配条件的值的表达式。
- ngSwitchDefault, 标记默认元素的属性。



注意不要在 ngSwitch 前使用 “*”, 而应该用属性绑定。另外, 在 ngSwitchCase 和 ngSwitchDefault 的前面要加上 “*”。

例如, 根据组件的 contactName 属性来确定展示对应的用户中文名。示例代码如下:

```
<span [ngSwitch]="contactName">
  <span *ngSwitchCase="'TimCook'"> 蒂姆·库克 </span>
  <span *ngSwitchCase="'BillGates'"> 比尔·盖茨 </span>
  <span *ngSwitchDefault> 无名氏 </span>
</span>
```

在上面示例代码中, 每个子指令 ngSwitchCase 都根据 ngSwitch 属性绑定的条件值来进行相关匹配, 看是否符合某个子指令的判断条件, 若符合则把该元素添加到 DOM 树的节点上, 并移除其兄弟元素; 若匹配不到所有的条件值, 则显示子指令 ngSwitchDefault 对应的模板元素。

7.3.5 NgFor

通过 NgFor 指令可以实现重复执行某些步骤来展示数据, 例如用来展示多列的模板列表, 这些模板元素结构及布局一致, 只是展示的具体数据不一样。示例代码如下:

```
<li *ngFor="let contact of contacts">
  <list-item [contact]="contact" (routerNavigate)="routerNavigate($event)"></list-item>
</li>
```

赋值给 *ngFor 的字符串并不是一个模板表达式, 前面的星号不能省略, 这是 Angular 提供的一种语法糖。在上面例子中, Angular 会遍历出 contacts 对象数组中的每个 contact, 并把它存储在局部变量 contact 中, 使其在每个循环迭代中对模板 HTML 可用。

NgFor 中的索引

NgFor 指令支持一个可选的 `index` 索引，在循环迭代过程中，其下标范围是 $0 \leq \text{index} < \text{数组的长度}$ 。开发者可以通过模板输入变量来捕获这个 `index`，并应用在模板中。比如把 `index` 赋值给变量 `i` 后，在当前的元素及其子元素中都可以使用该变量。示例代码如下：

```
<div *ngFor="let contact of contacts; let i=index">{{i + 1}} - {{ contact.id }}</div>
```

NgForTrackBy

在一些包含复杂列表的项目中，每次更改都会引发很多相互关联的 DOM 操作，这里使用 NgFor 指令会让性能变得很差。在通讯录例子中，当重新从服务器拉取列表数据时，拉取到的数据可能包含很多（可能不是全部）之前显示过的数据。虽然这些数据中的联系人编号（`contact.id`）并没有发生变化，但 Angular 并不知道哪些列表数据在数据更新前已经渲染过，只能清理旧列表的 DOM 元素，并用新的列表数据填充 DOM 元素来重建一个新列表。

在这种情况下，可以通过追踪函数来避免这种重复渲染的性能浪费。追踪函数可以让 Angular 将具有相同 `id` 的对象处理成同一个联系人。示例代码如下：

```
trackByContacts(index: number, contact: Contact) {  
    return contact.id;  
}
```

然后，通过 NgForTrackBy 指令设置追踪函数：

```
<div *ngFor="let contact of contacts; trackBy: trackByContacts">{{contact.id}}</div>
```

如果检查出同一个联系人的属性发生了变化，Angular 就会更新 DOM 元素；反之，就会留下这个 DOM 元素。使用 NgForTrackBy 指令的最终效果是列表界面变得更顺畅、响应更及时。

7.4 表单

前面部分讲解了如何使用模板完成数据绑定，本节将讲解在 Angular 中如何使用模板表单。

表单的使用场景非常广泛，常见的场景有用户注册、用户登录、数据的添加和修改、问卷调查、文件上传等。虽然 HTML 内置了表单标签，但它的一些标签特性存在浏览器兼容问题，并且自定义校验规则及表单数据获取、处理、提交等流程比较复杂。

针对上述问题，Angular 团队对表单进行了封装扩展，提供了很好的解决方案。Angular 表单提供了双向的数据绑定、强大的校验规则及自定义校验错误提示等功能，使得开发者可以使用简洁的代码、灵活的接口来构建功能强大的表单。

Angular 提供了模板驱动（Template-Driven Form）和模型驱动（Model-Driven Form）两种方式来构建表单。模板驱动模式使用模板表单内置指令、内置校验的方式来构建表单；模型驱动模式采用自定义表单、自定义校验的方式来构建表单。

本节将重点学习模板驱动方式构建表单，接着学习自定义表单校验，同时会引入模型驱动方式构建表单的内容。

7.4.1 模板表单例子

在深入理解表单相关知识之前，可以通过通讯录添加联系人的例子来简单了解模板中的表单，效果如图 7-2 所示。

图 7-2 添加联系人表单效果图

在添加联系人的表单中，需收集联系人姓名、电话、住址、邮箱、生日的信息，在模板中加入表单及其控件元素，创建文件 `form.component.ts`。示例代码如下：

```
@Component({
  selector: 'add-contact',
  template: `
<h3> 添加联系人 </h3>
<form>
  <ul>
    <li>
      <label for="name"> 姓名: </label>
      <input type="text" name="name"/>
    </li>
    <!-- ... -->
    <li>
      <button type="submit"> 添加 </button>
      <button type="button"> 取消 </button>
    </li>
  </ul>
</form>
`,
})
export class FormComponent {}
```

如上例所示，在 Angular 模板中构建表单与在普通 HTML 中构建表单类似，这里构建的模板表单只实现了一个简单的表单视图，并未添加任何交互处理。在 Angular 中，表单的交互是由表单的特有指令实现的，下面将通过表单指令进一步讲解 Angular 的表单。

7.4.2 表单指令

Angular 对常用的表单交互功能进行了封装扩展，形成了表单指令。其目的是负责处理数据绑定、指定校验规则、显示校验错误信息等，最终使开发者能在模板中快速构建交互友好的表单。接下来会一一讲解各个表单指令的功能，以及由表单指令引申出的相关知识点，涉及的内容如图 7-3 所示。

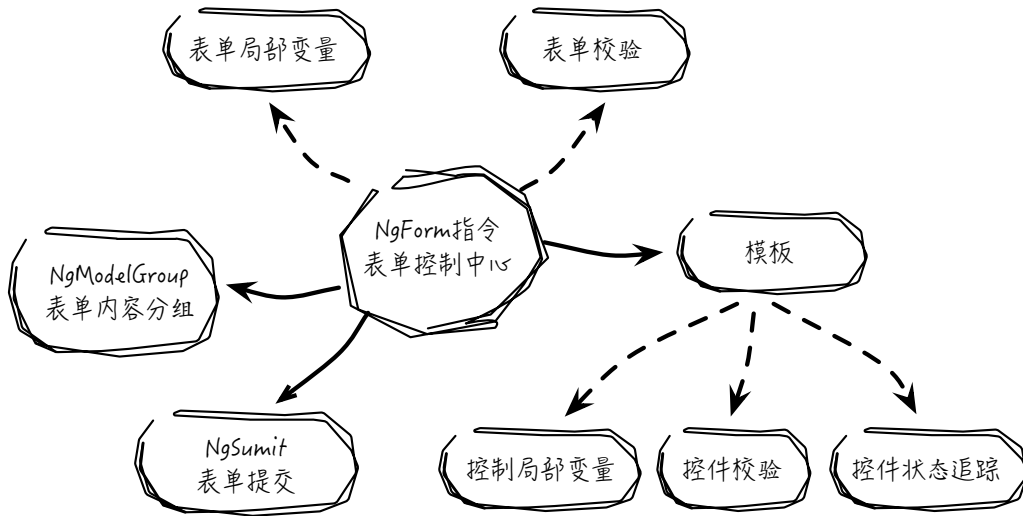


图 7-3 表单指令内容

NgForm 指令

NgForm 指令是表单的控制中心，负责处理表单内的页面逻辑，为普通的表单元素扩充了许多额外的特性，所有的表单指令都只有在 NgForm 指令内部才能正常运行。先来看看在模板中如何使用 NgForm 指令。

首先，在根模块中添加如下代码：

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { FormComponent } from './form.component';
```

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule // 导入FormsModule
  ],
  declarations: [
    AppComponent,
    FormComponent
  ],
```

```
bootstrap: [AppComponent]
})
export class AppModule { }
```

在上面代码中，导入了 `FormsModule` 模块和 `FormComponent` 组件。将 `FormComponent` 组件添加到 `@NgModule` 元数据的 `declarations` 数组中，其目的是在整个模块中都可以使用 `FormComponent` 组件。将 `FormsModule` 模块添加到 `@NgModule` 元数据的 `imports` 数组中，这使得在整个应用的模板驱动表单中都可以使用特有的表单指令。

接下来，就可以在 `FormComponent` 组件的模板中直接使用 `NgForm` 指令了。开发者可以不用在模板中显式使用 `NgForm` 指令，因为添加了 `FormsModule` 模块后，Angular 模板在编译解析时，遇到 `<form>` 标签会自动创建一个 `NgForm` 指令并且将其添加到该 `<form>` 标签上。

`NgForm` 指令控制通过 `NgModel` 指令和 `name` 属性创建的控件类，并且也会跟踪控件类的属性变化，包括有效性属性（`valid`）。本章后续内容还会结合其他指令来详细讲解 `NgForm` 指令。

NgModel 指令

`NgModel` 指令是表单数据绑定的核心所在，是表单运用中最重要的一个指令，几乎所有的表单特性都依赖 `NgModel` 指令实现。`NgModel` 指令实现了表单控件的数据绑定，提供了控件状态跟踪及校验功能。Angular 表单支持单向和双向数据绑定，表单的单向数据绑定使用 `[ngModel]`，双向数据绑定使用 `[(ngModel)]`。示例代码如下：

```
<input type="text" name="contactName" [ngModel]="curContact.name" >
<input type="text" name="contactName" [(ngModel)]="curContact.name">
```

上例中为控件添加了 `[(ngModel)]` 属性绑定，当在视图文本控件中输入文本时，组件中数据 `curContact.name` 会被更新。当组件中数据 `curContact.name` 发生变化时，在视图文本控件中显示的内容也会被同步更新。

在控件中使用 `NgModel` 属性绑定，必须给该控件添加 `name` 属性，否则会报错。因为 `NgForm` 指令会为表单建立一个控件对象 `FormControl` 的集合，以此来作为表单控件的容器。控件的 `NgModel` 属性绑定会以 `name` 作为唯一标识符来注册并生成一个 `FormControl`，将其加入 `FormControl` 的集合中。



在本章 7.2 节中，已经详细讲解了单向、双向数据绑定的用法及原理，这里不再赘述。

表单中的一些控件可以使用 `NgModel` 指令来实现双向数据绑定，如上例中设置文本控件来实现双向数据绑定。接下来将介绍一些常用的表单控件。

单选钮

单选钮控件（Radio）实现双向数据绑定，同一组单选钮控件的所有 `[(ngModel)]` 属性都必须绑定同一个模型数据，且 `name` 属性名也必须相同。示例代码如下：

```
<input type="radio" name="sex" [(ngModel)]="curContact.sex" value="female"> 女  
<input type="radio" name="sex" [(ngModel)]="curContact.sex" value="male"> 男
```

上面例子实现了一组选择性别单选钮，两个单选钮的 `NgModel` 指令绑定的是一个模板表达式 `curContact.sex`，当选中选项为“女”的单选钮时，`curContact.sex` 会被赋值为 `female`；若选中“男”单选钮，则所对应的值为 `male`。

复选框

复选框控件（Checkbox）用来表示该表单复选框是否被选中，其中 `[(ngModel)]` 属性绑定的是一个布尔值。示例代码如下：

```
<input type="checkbox" name="lock" [(ngModel)]="curContact.lock"> 禁用
```

在上面例子中，如果复选框被选中，则 `curContact.lock` 的值为 `true`，否则为 `false`。

单选下拉框

单选下拉框控件（Select）的双向数据绑定，需结合 `option` 元素绑定的值来实现。`option` 选项的元素属性的绑定目标有两种，分别为 `value` 和 `ngValue`。当在 `option` 元素中使用 `value` 绑定数据时，其返回值类型是基本数据类型；当使用 `ngValue` 绑定数据时，其返回值类型是对象数据类型。

在构建下拉框前，需要先定义下拉框列表所需的数据。示例代码如下：

```
export class FormComponent {  
  
  interests:any[] = [  
    {value: 'reading', display: '阅读'},  
    {value: 'traveling', display: '旅游'},  
    {value: 'sport', display: '运动'}  
  ];  
  
  // ...  
}
```

接下来实现一个单选钮被选中后返回基本类型数据的例子。示例代码如下：

```
<select name="interestValue" [(ngModel)]="curContact.interestValue">
  <option *ngFor="let interest of interests" [value]="interest.value">
    {{interest.display}}
  </option>
</select>
```

在上面例子中，使用 [value] 来绑定下拉选项的 value 属性值，如果选中下拉选项中的“旅游”，那么 curContact.interestValue 的值将变为 traveling。

单选钮被选中后也可以返回对象类型数据。示例代码如下：

```
<select name="interestObj" [(ngModel)]="curContact.interestObj">
  <option *ngFor="let interest of interests" [ngValue]="interest">
    {{interest.display}}
  </option>
</select>
```

在上面例子中，使用 [ngValue] 来绑定下拉选项的 value 属性值，当选中某个选项后，该下拉框最终返回一个 Object 类型的对象数据。如选中下拉选项中的“旅游”，则 curContact.interestObj 的值为 { value: 'traveling', display: '旅游' }。

多选下拉框

多选下拉框控件（Multiple Select）实现了下拉选择多个选项的功能。多选下拉框的用法与单选下拉框类似，不同的是多选下拉框返回的数据是一个由所有被选择数据组成的数组。

接下来实现一个多选下拉框，返回一个成员为字符串的数组。示例代码如下：

```
<select multiple name="interestMul" [(ngModel)]="curContact.interestMul">
  <option *ngFor="let interest of interests" [value]="interest.value">
    {{interest.display}}
  </option>
</select>
```

在上面例子中，多选下拉框使用 [value] 来绑定下拉选项的 value 属性值，如果在多选下拉框中选中“旅游”和“运动”两项，则 curContact.interestMul 返回的值为 ["traveling", "sport"]。

多选下拉框还可以使用 [ngValue] 来绑定下拉选项的 value 属性值，如果在多选下拉框中选中“旅游”和“运动”两项，那么 curContact.interestMul 返回的数组对象为

```
[{ value: 'traveling', display: '旅游' }, { value: 'sport', display: '运动' } ] ]。
```

模板局部变量

模板局部变量（Template Reference Variable，简称局部变量）是模板中对 DOM 元素或指令（包括组件）的引用，可以使用在当前元素、兄弟元素或任何子元素中。

DOM 元素局部变量

若想在标签元素中定义 DOM 元素局部变量，那么只需在局部变量名前面加上 “#” 符号，或者用 “ref-” 前缀即可。示例代码如下：

```
<li>
  <label for="name"> 姓名: </label>
  <input type="text" #contactName name="contactName" id="contactName">
  <input type="number" ref-telNum name="telNum" id="telNum">
  <p>{{contactName.value}} -- {{telNum.value}}</p>
</li>
```

Angular 会自动把局部变量设置为对当前 DOM 元素对象的引用，如上面例子中的局部变量 `contactName` 引用的是 `document.getElementById("contactName")` 对象。在模板中定义局部变量后，可以直接在模板的其他元素中使用该元素的 DOM 属性，如上面例子中的 `contactName.value` 和 `telNum.value`。

表单指令局部变量

表单指令也可以定义局部变量，其引用方式与 DOM 元素局部变量的引用方式不同。表单指令的局部变量在定义时需手动初始化为特定指令的代表值，最终解析后会被赋值为表单指令实例对象的引用。

NgForm 表单局部变量

如下面的例子，在表单中定义局部变量 `contactForm`，将 `contactForm` 变量初始化为 `ngForm`，并在表单控件中加入 `ngModel` 及 `contactName` 属性。示例代码如下：

```
<form #contactForm="ngForm">
  <ul>
    <li>
      <label for="contactName"> 姓名: </label>
      <input type="text" name="contactName" [(ngModel)]="curContact.name" >
    </li>
  </ul>
</form>
```

```

    <label for="telNum"> 电话: </label>
    <input type="text" name="telNum" [(ngModel)]="curContact.telNum">
  </li>
  ...
</ul>
</form>

```

局部变量 `contactForm` 是对 `NgForm` 指令实例对象的引用，可以在模板中读取 `NgForm` 实例对象的属性值，如追踪表单的 `valid` 属性状态。当被包含的所有控件都有效时，`contactForm.valid` 的值为 `true`，否则为 `false`。在控件中添加 `ngModel` 和 `name` 属性后，若在“姓名”控件中输入“李四”，在“电话”控件中输入“123456789”，则 `contactForm.value` 的值为：

```

{
  contactName: '李四',
  telNum: '123456789'
}

```

`contactForm` 对象的 `value` 属性是一个简单的 JSON 对象，该对象的键对应控件元素的 `name` 属性值，而其值对应控件元素的 `value` 值。

NgModel 控件局部变量

下面的例子实现了一个文本控件，将 `[(ngModel)]` 初始化为联系人姓名，并添加了控件局部变量 `name`。示例代码如下：

```

<input type="text" name="contactName" [(ngModel)]="curContact.name" #contactName="
  ngModel">
<p>{{contactName.valid}}</p>

```

局部变量 `contactName` 是对 `NgModel` 指令实例对象的引用，可以在模板中读取 `NgModel` 实例对象的属性值，如通过 `contactName.valid` 可以追踪控件状态、表单校验不通过时提示错误信息等。

Angular 提供的 `NgForm` 表单局部变量和 `NgModel` 控件局部变量，在模板中为追踪表单状态及进行表单数据校验提供了便利。

表单状态

表单指令 `NgForm` 和 `NgModel` 都可以用于追踪表单状态来实现数据校验。表单指令 `NgForm` 和 `NgModel` 都有 5 个表示状态的属性，属性值都为布尔类型，并且都可以通过

对应的局部变量获取。其中 `NgForm` 追踪的是整个表单控件的状态，`NgModel` 追踪的是其所在表单控件的状态。表单状态的属性语义如表 7-4 所示。

表 7-4 表单状态的属性语义

状态	true / false
valid	表单值是否有效
pristine	表单值是否未改变
dirty	表单值是否已改变
touched	表单是否已被访问过
untouched	表单是否未被访问过

我们以添加姓名的表单控件为例，首先给该控件添加 `required` 属性，然后获取焦点并输入姓名，最后移除焦点，通过这些步骤来观察每一步操作完成后表单控件的状态变化，如图 7-4 所示。

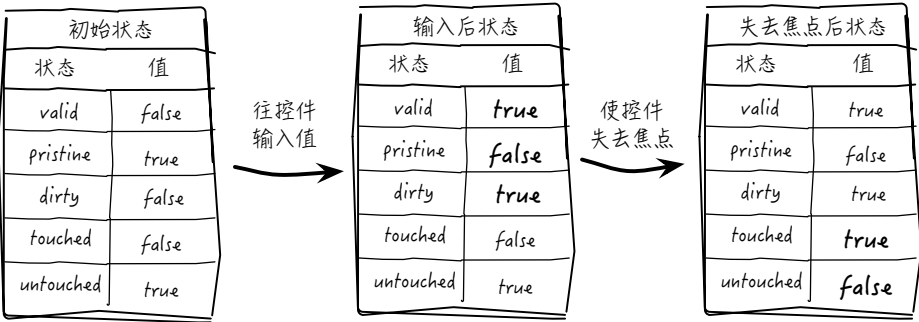


图 7-4 表单控件的状态变化

由此可见，用户操作会改变表单的属性状态，所以可以通过检查当前的属性状态值来赋予表单特定的样式或加入特定的处理逻辑。

NgModelGroup 指令

`NgModelGroup` 是 Angular 提供的另一个特色指令，可以对表单输入内容进行分组，方便我们在语义上区分不同类型的输入。例如联系人的信息包括姓名和住址，对这两者还可以进行更精细化的信息收集，如“姓名”可细分为“姓”和“名字”，“地址”可细

分为“城市”“区”“街道”等。通过 NgModelGroup 可以将姓名和住址进行分组收集，示例代码如下：

```
<form #concatForm="ngForm">
  <fieldset ngModelGroup="nameGroup" #nameGroup="ngModelGroup">
    <label> 姓:</label>
    <input type="text" name="firstname" [(ngModel)]="curContact.firstname" required
      >
    <label> 名字:</label>
    <input type="text" name="lastname" [(ngModel)]="curContact.lastname" required>
  </fieldset>
  <fieldset ngModelGroup="addressGroup" #addressGroup="ngModelGroup">
    <label> 街道:</label>
    <input type="text" name="street" [(ngModel)]="curContact.street" required>
    <label> 区:</label>
    <input type="text" name="zip" [(ngModel)]="curContact.zip" required>
    <label> 城市:</label>
    <input type="text" name="city" [(ngModel)]="curContact.city" required>
  </fieldset>
</form>
```

上面例子中分别对联系人的姓名和住址进行了分组，通过 ngModelGroup 指令，将姓和名字的表单内容包裹成姓名分组，用 nameGroup 表示；将城市、区和街道的表单内容包裹成住址分组，用 addressGroup 表示。此时 concatForm.value 的值为：

```
{
  nameGroup: {
    firstname: '',
    lastname: ''
  },
  addressGroup: {
    street: '',
    zip: '',
    city: ''
  }
}
```

除此之外，使用 NgModelGroup 实例对象的 valid 属性可以单独校验其所在分组控件的输入是否有效。如在上例的姓名分组中，只有当 curContact.firstname、curContact.lastname 输入都有效时，局部变量 nameGroup.valid 的值才会变为 true，否则为 false。

ngSubmit 事件

ngSubmit 事件可以响应表单里类型为 submit 的按钮操作，并负责控制表单的提交流程。当按钮被点击后，会触发表单的 ngSubmit 事件。下面我们通过一个表单提交的例子来学习 ngSubmit 事件。组件模板的示例代码如下：

```
<form #contactForm="ngForm" (ngSubmit)="doSubmit(contactForm.value)">

  <!-- ... -->

  <li class="form-group">
    <button type="submit" class="btn btn-default" [disabled]="!contactForm.valid">
      添加</button>
    <button type="reset" class="btn btn-default">重置</button>
  </li>
</form>
```

提交按钮的处理逻辑的示例代码如下：

```
export class FormComponent {
  doSubmit(formValue: any){
    // 处理表单数据并提交
  }
}
```

在上面例子中，绑定了 ngSubmit 事件，它的类型是 EventEmitter。当提交按钮被点击后，首先执行表单原生的 onSubmit 事件，然后执行 FormComponent 组件中定义的 doSubmit() 方法，该方法接收 contactForm.value 的值作为参数传入，并对传入的数据进行处理。

上述代码中添加了 [disabled]="!contactForm.valid" 来绑定提交按钮的 disabled 属性，用于验证表单的可提交状态，当表单的输入值全部校验通过后，contactForm.valid 的值为 true。在表单中还添加了类型为“reset”的按钮，该按钮实现了 Angular 表单重置的功能，点击该按钮会执行表单原生的 onReset 事件并触发表单内容的重置。

7.4.3 自定义表单样式

NgModel 指令不仅仅能追踪表单控件的状态，还会根据表单控件的状态使用对应的 CSS 状态类来更新表单控件的类名。表单控件包括 6 个 CSS 状态类，具体如表 7-5 所示。

表 7-5 表单控件的 CSS 状态类

状态	为 true 时的 CSS 状态类	为 false 时的 CSS 状态类
控件是否已经被访问过	ng-touched	ng-untouched
控件值是否已经变化	ng-dirty	ng-pristine
控件值是否有效	ng-valid	ng-invalid

表单控件的 CSS 类名会根据表单控件状态的变化而变化。在实际的应用场景中，当控件状态变化时，如在无效状态与有效状态的切换过程中，需要在视觉上进行区分，那么使用表单控件的 CSS 状态类就能方便地实现视觉切换效果。示例代码如下：

```
.ng-valid[required] {  
  border-left: 5px solid #0f0; /* 绿色 */  
}  
.ng-invalid {  
  border-left: 5px solid #f00; /* 红色 */  
}
```

当输入有效时控件边框变为绿色，当输入无效时则变为红色。此外，还可以利用其他表单的 CSS 状态类自定义功能更加强大的控件。但这样还不够，比如用户无法得知输入为什么不合法，以及纠正输入值等。在这种场景下可结合控件的状态属性 valid、pristine、dirty、touched、untouched 等来添加有用的消息提示，以此来对用户进行有效的引导。示例代码如下：

```
<div class="form-group">  
  <label for="contactName"> 姓名 </label>  
  <input type="text" class="form-control" minlength=3 maxlength=10 [(ngModel)]="curContact.name" name="contactName" #contactName="ngModel" required >  
    <p [hidden]="contactName.valid || contactName.pristine" class="alert alert-invalid">  
      用户名长度为 3~10 个字符  
    </p>  
</div>
```

结合刚刚提到的样式示例，当表单在初始状态或者输入值有效时，控件边框为绿色，不显示错误提示；当表单输入无效时，控件边框为红色，并显示错误提示。

7.4.4 表单校验

表单校验（Validation）用来检查表单的输入值是否满足所设定的规则，如果不满足，则将相关状态立即反馈给用户。虽然 HTML 5 表单内置了相关的基础校验，但这些基础校验的使用场景有限，且各个浏览器的兼容性相差较大。Angular 封装了相关的表单校验规则，并提供了灵活的接口，以便能够高效地完成表单校验。本节将详细介绍 Angular 表单的内置校验及自定义校验。

表单内置校验

Angular 支持的表单内置校验（Built-In Validator）包括：

- required，判断表单控件值是否为空。
- minlength，判断表单控件值的最小长度。
- maxlength，判断表单控件值的最大长度。
- pattern，判断表单控件值的匹配规则。

使用 Angular 表单内置校验与使用普通的 HTML 校验一致，直接在表单控件中添加对应的校验属性即可。示例代码如下：

```
<input type="text" minlength=3 maxlength=10 [(ngModel)]="curContact.name" name="contactName" required />
```



HTML 支持简单的拦截校验，但是其提示样式及文本是固定且不可控的，当引入了 FormsModule 后，Angular 会自动在 <form> 标签中添加 novalidate 属性来屏蔽 HTML 校验。如果需要使用 HTML 原生的表单校验，则可在 <form> 标签上使用 ngNoForm 或者 ngNativeValidate。

表单自定义校验

Angular 提供的表单内置校验基本能满足大部分业务场景的校验需求，如果需要实现复杂的表单校验功能，则可以使用 Angular 提供的表单自定义校验（Custom Validator）。

创建自定义校验

下面自定义一个用户名的校验器，校验规则为：用户名必须是邮箱、手机号码的格式。示例代码如下：

```
// validate-username.ts
```

```
import { FormControl } from '@angular/forms';
const EMAIL_REGEXP = new RegExp("[a-z0-9]+@[a-z0-9]+.com");
const TEL_REGEXP = new RegExp("1[0-9]{10}");
export function validateUserName(c: FormControl) {
  return (EMAIL_REGEXP.test(c.value) || TEL_REGEXP.test(c.value)) ? null : {
    userName: {
      valid: false,
      errorMsg: '用户名必需是手机号或者邮箱账号'
    }
  };
}
```

自定义校验函数传入的参数是 `FormControl` 类，通过对 `FormControl` 的 `value` 值进行校验处理，返回校验结果。

使用自定义校验

本节我们只介绍在使用模型驱动方式构建的表单中如何使用自定义校验。使用模型驱动方式构建表单，需要在表单组件所在的模块代码中导入 `ReactiveFormsModule`，并在模块的 `@NgModule` 元数据的 `imports` 数组中加入 `ReactiveFormsModule`。示例代码如下：

```
// ...
import { ReactiveFormsModule } from '@angular/forms';
import { FormComponent } from './form.component';
import { AppComponent } from './app.component';

@NgModule({
  imports: [BrowserModule, ReactiveFormsModule],
  declarations: [AppComponent, FormComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

导入 `ReactiveFormsModule` 后，可在表单组件 `FormComponent` 中使用模型驱动方式构建表单。构建表单组件及使用自定义校验的代码如下：

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
import { validateUserName } from './validate-username';
// 本节开头创建的自定义校验器
```

```
@Component({
  selector: 'add-contact',
  template: `
    <form [formGroup]="customForm">
      <label> 姓名:</label>
      <input type="text" formControlName="customName">
    </form>
  `,
})
export class FormComponent {
  customForm = new FormGroup({
    customName: new FormControl('', validateUserName)
  });
}
```

该例子定义了 `customForm`（即 `FormGroup` 表单实例对象）和 `customName`（即 `FormControl` 控件实例对象）。在构建 `FormControl` 实例对象 `customName` 时传入的参数中，第一个参数为控件返回值的初始值，第二个参数为该控件的校验配置方法。

此外，校验配置可以使用 `Validators` 内置校验，如 `Validators.required()`、`Validators.minLength()` 等，这与直接在表单控件元素中添加 `required`、`minlength` 属性效果是一致的。使用 `Validators` 内置校验，需先从 `@angular/forms` 导入 `Validators`。示例代码如下：

```
import { Validators } from '@angular/forms';

// ...
customForm = new FormGroup({
  customName: new FormControl('', Validators.minLength(4))
});
// ...
```

如果需要在 一个表单中添加多个校验器，则可以在校验配置参数中使用数组，数组元素为对应的校验方法。示例代码如下：

```
// ...
customForm = new FormGroup({
  customName: new FormControl('', [Validators.minLength(4), validateUserName]),
});
// ...
```

验证时机控制

在默认情况下，当用户进行输入时便会执行验证操作，即触发 input 事件。这样的设计满足大部分应用场景，但有一些特殊场景，如异步耗时的验证等，频繁地触发验证并不友好。好在自 Angular 5.0 之后表单支持修改验证触发的事件，如改成 blur 事件。示例代码如下：

```
<!-- 失去焦点时才触发控件校验 -->
<input name="contactName" [(ngModel)]="curContact.name" [ngModelOptions]="{updateOn: 'blur'}" required >
```

也可以更改整个表单的验证时机，示例代码如下：

```
<!-- 当表单的某个控件失去焦点时才触发表单校验 -->
<form [ngFormOptions]="{updateOn: 'blur'}">
<!-- 当表单提交时才触发校验 -->
<form [ngFormOptions]="{updateOn: 'submit'}">
```

7.5 管道

在 Angular 中，管道（Pipe）可以按照开发者指定的规则对模板内的数据进行转换。如日期类型（Date），默认会显示为 Mon Jun 06 2016 14:17:00 GMT+0800 (CST) 这样的形式，但不如 Jun 6, 2016 这样的形式直观，此时可以借助于管道来实现这种转换效果。本节将学习管道的基本用法、内置管道、自定义管道、纯管道及非纯管道等内容。

7.5.1 管道介绍

使用管道，需要用管道操作符“|”来连接模板表达式中左边的输入数据和右边的管道。示例代码如下：

```
@Component({
  selector: 'pipe-demo',
  template: `
    <p>My birthday is {{ birthday | date }}</p>
  `,
})
export class PipeDemoComponent {
  birthday = new Date(1999, 3, 15);
}
```

在上面例子中，通过管道操作符右边的 `date` 管道来实现日期数据的格式转换，输出结果为 `My birthday is Apr 15, 1999`。`date` 管道是 Angular 的内置管道，存放在 `CommonModule` 里，下文会展开讲述。

管道参数

管道可以使用参数，通过传入的参数来输出不同格式的数据。如日期需要以固定格式输出，可以使用 “:” 给日期管道添加参数。示例代码如下：

```
<p>My birthday is {{ birthday | date:"MM/dd/y" }}</p>
```

上面的例子最终在页面中展示为：My birthday is 04/15/1999。

链式管道

一个模板表达式可以连续使用多个管道进行不同的处理，这就是链式管道。使用链式管道可以展示更丰富的数据格式，其语法格式如下：

```
{{ expression | pipeName1 | pipeName2 | ... }}
```

模板表达式 `expression` 的值通过管道 `pipeName1` 处理后再传递给 `pipeName2` 管道处理，直至最后一个管道处理完成后，就可以输出链式管道处理的最终结果了。

7.5.2 内置管道

Angular 根据业务场景封装了一些常用的内置管道。内置管道可以直接在任何模板表达式中使用，不需要通过 `import` 导入和在模块中声明。Angular 提供的常用的内置管道如图 7-5 所示。

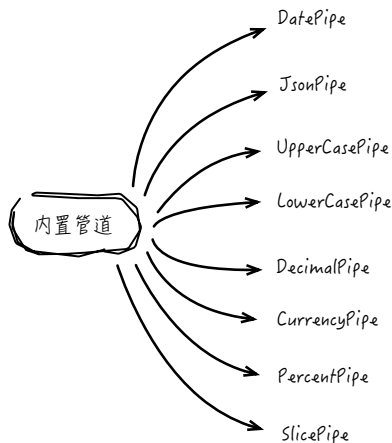


图 7-5 常用的内置管道

这些内置管道给我们提供了较不错的功能，它们的类型及功能如表 7-6 所示。

表 7-6 常用内置管道的类型及功能

管道	类型	功能
DatePipe	纯管道	日期管道，格式化日期
JsonPipe	非纯管道	将输入数据对象经过 JSON.stringify() 方法转换后输出对象字符串
UpperCasePipe	纯管道	将文本中所有小写字母转换成大写字母
LowerCasePipe	纯管道	将文本中所有大写字母转换成小写字母
DecimalPipe	纯管道	将数值按特定的格式显示为文本
CurrencyPipe	纯管道	将数值转换成本地货币格式
PercentPipe	纯管道	将数值转换成百分比格式
SlicePipe	非纯管道	将数组或者字符串裁剪成新子集

DatePipe

DatePipe 管道用来格式化日期数据，其使用格式如下：

expression | date: format

其中，expression 可以为 Date 日期对象、日期字符串，如“2016/04/05”或者毫秒级的时间戳。format 为自定义的日期格式，Angular 提供了年、月、日等标识符，可以根据标识符来自定义日期格式。这些标识符如表 7-7 所示。

表 7-7 DatePipe 管道日期标识符（以 2016-06-08 20:05:08 时间为例）

日期	标识符	缩写	全称	单标识符	双标识符
地区	G	G (AD)	GGGG (Anno Domini)	—	—
年	y	—	—	y (2016)	yy (16)
月	M	MMM (Jun)	MMMM (June)	M (6)	MM (06)
日	d	—	—	d (8)	dd (08)
星期	E	EEE (Fri)	EEEE (Friday)	—	—
时间 (AM, PM)	j	—	—	j (8 PM)	jj (08 PM)
12 小时制时间	h	—	—	h (8)	hh (08)
24 小时制时间	H	—	—	H (20)	HH (20)
分	m	—	—	m (5)	mm (05)

续表

日期	标识符	缩写	全称	单标识符	双标识符
秒	s	—	—	s (8)	ss (08)
时区	Z	—	Z (china Standard Time)	—	—
时区	z	z (GMT-8:00)	—	—	—

接下来通过具体例子来学习如何使用这个内置管道。示例代码如下：

```
@Component({
  selector: 'pipe-demo-date',
  template: `
    <p>{{date | date: "y-MM-dd EEEE"}}</p>
  `
})
export class PipeDemoDateComponent {
  date: Date = new Date('2016-06-08 20:05:08');
}
```

输出结果为：

2016-06-08 Wednesday

JsonPipe

JsonPipe 管道通过 JSON.stringify() 将输入数据对象转换成对象字符串，该管道主要用于开发调试。示例代码如下：

```
@Component({
  selector: 'pipe-demo-json',
  template: `
    <pre>{{jsonObject | json}}</pre>
  `
})
export class PipeDemoJsonComponent {
  jsonObject: Object = {foo: 'bar', baz: 'qux', nested: {xyz: 3, numbers: [1, 2]}};
}
```

输出结果为：

```
{
  "foo": "bar",
  "baz": "qux",
  "nested": {
    "xyz": 3,
    "numbers": [
      1,
      2
    ]
  }
}
```

UpperCasePipe

UpperCasePipe 管道用于将文本中所有小写字母转换成大写字母，其语法格式如下：

expression | uppercase

LowerCasePipe

LowerCasePipe 管道用于将文本中所有大写字母转换成小写字母，其语法格式如下：

expression | lowercase

DecimalPipe

DecimalPipe 管道用于对数值的整数与小数部分按照指定规则进行格式化，这种格式化方式也称为本地格式化处理，其语法格式如下：

expression | number[: digitInfo]

参数 digitInfo 的格式如下：

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

- minIntegerDigits: 整数部分保留最小的位数，默认值为 1。
- minFractionDigits: 小数部分保留最小的位数，默认值为 0。
- maxFractionDigits: 小数部分保留最大的位数，默认值为 3。

具体用法的示例代码如下：

```
@Component({
  selector: 'pipe-demo-number',
```

```

template: `
  <div>
    <p>a 变量: {{a | number: '3.4-5'}}</p>
    <p>b 变量: {{b | number: '3.1-5'}}</p>
  </div>
`,
})
export class PipeDemoNumberComponent {
  a: number = 2.718281828459045;
  b: number = 33456;
}

```

转换后的结果为:

```

a 变量: 002.71828
b 变量: 33,456.0

```

格式化变量 a 所采用的参数为 3.4-5，参数“.”左边的 3 表示整数位最少保留 3 位，原数值的整数位为 1 位，因不足 3 位，采用前导 0 进行填充，填充后整数位变为 002。参数“.”右边的 4-5 表示保留小数的最小位数为 4 位，最大位数为 5 位，因原数值的小数位大于 5 位，故四舍五入后保留 5 位小数。整合整数位和小数位后，最终的转换结果是 002.71828。

格式化变量 b 所采用的参数为 3.1-5，参数“.”左边为 3，因原数值的整数位为 5 位，大于 3 位，直接将整数位格式化为 33,456（注意这里的整数部分，会从右往左每三位额外增加一个逗号“,”）。参数“.”右边为 1-5，因原数值没有小数位，因此采用最小保留 1 位限制的规则，小数位补一个默认值 0。整合整数位和小数位后，最终的转换结果是 33,456.0。

CurrencyPipe

CurrencyPipe 管道可以对数值进行本地货币格式化处理，其语法格式如下：

```
expression | currency[: currencyCode[: symbolDisplay[: digitInfo]]]
```

- 参数 currencyCode 表示要格式化的目标货币格式，其值为 ISO 4217 货币码，如 CNY 为人民币、USD 为美元、EUR 为欧元等。
- 参数 symbolDisplay 表示以该类型货币的哪种格式显示，其值为布尔值，true 表示显示货币符号如 ¥、\$ 等，false 则表示显示 ISO 4217 货币码如 CNY、USD 等。
- digitInfo 详情可参考上面介绍的 DecimalPipe 管道的 digitInfo 格式说明。

示例代码如下：

```
@Component({
  selector: 'pipe-demo-currency',
  template: `<div>
    <p>A 变量: {{ a | currency: 'USD': false }}</p>
    <p>B 变量: {{ b | currency: 'USD': true: '4.2-2' }}</p>
  </div>`
})
export class PipeDemoCurrencyComponent {
  a: number = 0.259;
  b: number = 1.3495;
}
```

转换后的结果为：

A 变量: USD0.259

B 变量: \$0,001.35

在上面例子中，变量 a 的第一个参数为 USD，表示转换成美元货币格式；第二个参数为 false，表示将货币前缀展示成 ISO 4217 货币码中的美元 USD。因此，最终解析后转换为 USD0.259。变量 b 的第一个参数与 a 相同；第二个参数为 true，表示货币前缀需展示为美元货币符号 \$；第三个参数表示对货币数值进行格式化。因此，最终解析后转换为 \$0,001.35。

PercentPipe

PercentPipe 管道可以对数值进行本地百分比格式化处理，其语法格式如下：

```
expression | percent[: digitInfo]
```

其中，参数 digitInfo 可参考 DecimalPipe 管道的 digitInfo 格式说明。示例代码如下：

```
@Component({
  selector: 'pipe-demo-percent',
  template: `
    <div>
      <p>A 变量: {{a | percent}}</p>
      <p>B 变量: {{b | percent:'4.3-5'}}</p>
    </div>
  `
})
```

```
export class PipeDemoPercentComponent{
  a: number = 0.259;
  b: number = 1.3495;
}
```

转换后的结果为:

A 变量: 25.9%

B 变量: 0,134.950%

SlicePipe

SlicePipe 管道用于裁剪数组或者字符串,并返回裁剪后的目标子集,其语法格式如下:

```
expression | slice: start[: end]
```

SlicePipe 的裁剪功能是基于 `Array.prototype.slice()` 和 `String.prototype.slice()` 方法来实现的。输入值 `expression` 可为数组或者字符串,参数 `start` 和 `end` 为相关的索引,具体定义可参考 `Array.prototype.slice()` 和 `String.prototype.slice()` 方法的介绍,这里不再赘述。

7.5.3 自定义管道

虽然 Angular 提供了许多内置管道,但是数据转换涉及各种各样的格式,内置管道显然无法满足全部需求,因此我们需要使用 Angular 提供的自定义管道功能来实现更多的需求。

比如在本章 7.4 节的单选钮双向数据绑定的例子中,假如服务器返回的性别数据是 `female` 或 `male`。而我們希望在页面中展示联系人性别时,看到的是中文字符的“女”或“男”。在这里就可以通过自定义管道将性别从英文字符转换成中文字符。接下来将通过自定义一个性别转换的管道来学习在模板中是如何自定义管道的。

定义元数据

在使用 `@Pipe` 定义元数据前必须从 `@angular/core` 中引入 `Pipe` 和 `PipeTransform`。示例代码如下:

```
// sexreform.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({name: 'sexReform'})
export class SexReform implements PipeTransform {
  // ...
}
```

通过 `@Pipe` 装饰器来告诉 Angular 这是一个管道类，`@Pipe` 的元数据有一个 `name` 属性，用来指定管道名称，这个名称必须是有效的 JavaScript 标识符，这里将管道命名为 `sexReform`。

实现 transform 方法

自定义管道必须继承接口类 `PipeTransform`，同时自定义管道必须实现 `PipeTransform` 接口的 `transform()` 方法，该方法的第一个参数为需要被转换的值，后面可以有若干个可选的转换参数，该方法需要返回一个转换后的值。下面实现一个性别转换的管道，示例代码如下：

```
// ...
export class SexReform implements PipeTransform {
  transform(val: string): string {
    switch(val) {
      case 'male': return '男';
      case 'female': return '女';
      default: return '未知性别';
    }
  }
}
```

使用自定义管道

在组件模板中使用自定义管道之前，必须在 `@NgModule` 的元数据 `declarations` 数组中添加自定义管道。示例代码如下：

```
import { SexReform } from 'pipes/sexreform.pipe';
// ...

@NgModule({
  // ...
  declarations: [SexReform]
})
```

添加到 `declarations` 数组后，就可以在模板中像内置管道一样使用自定义管道了。示例代码如下：

```
// ...
@Component({
  selector: 'pipe-demo-custom',
```

```
    template: `
    <p>{{sexValue | sexReform}}</p>
    `
  })
  // ...
```

7.5.4 管道的变化监测

Angular 在每次点击、移动鼠标、定时器触发及服务器响应等事件后都会对数据绑定进行变化监测，而频繁的变化监测则会引起性能问题。但是我们可以通过使用管道，让 Angular 选择使用更简单、更快速的变化监测策略来提高性能。下面通过使用管道实现一个过滤联系人列表功能的例子来了解 Angular 管道的性能优化。示例代码如下：

```
// ...
@Pipe({name: 'selectContact'})
export class SelectContactPipe implements PipeTransform {
  transform(allContacts: Array, prefix: string) {
    return allContacts.filter(contact => contact.name.match("^" + prefix));
  }
}

@Component({
  selector: 'pipe-demo',
  template: `
    <input type="text" #box (keyup.enter)="addContact(box.value); box.value=''"
      placeholder="输入联系人后按回车键添加">
    <div *ngFor="let contact of (contacts | selectContactPipe : '李')">
      {{contact.name}}
    </div>
  `
})
export class PipeDemoComponent {
  contacts=[{name:'张三'}, {name:'李四'}];
  addContact(name:string) {
    this.contacts.push({name});
  }
}
```

在上述代码中，定义了一个过滤联系人的管道 `SelectContactPipe`，并传入姓氏字符串“李”，联系人列表经过该管道的过滤，只显示姓“李”的联系人。如果用户在表单

中输入一个“李”姓联系人后，按回车键就会触发 `addContact()` 方法，并将新联系人加入数组中。此时我们预期在联系人列表中应该会实时显示新的“李”姓联系人，但结果并非如此。

因为 Angular 管道的变化监测策略对性能进行了优化，这种监测策略会忽略检查列表内部数据的变化。例子中使用 `this.contacts.push(contact)` 新增一个联系人，数组对象引用并没有发生改变。从 Angular 角度来说，引用地址不变的数组不进入 `SelectContactPipe` 筛选管道，所以列表数据并没有更新，页面不会实时显示更新后的联系人。这种筛选管道称为纯管道。虽然纯管道优化了性能，但是在上面例子的业务逻辑中，这样的优化却不符合我们真正的预期。要实现我们想要的功能，就需要 Angular 的另外一种变化监测机制了。

Angular 管道有两种变化监测机制，分别对应两种类型的管道，即纯管道和非纯管道。其中纯管道是默认类型。接下来看看纯管道和非纯管道的区别。

纯管道

在模板表达式中使用纯管道（Pure Pipe）后，只有在监测到输入值发生纯变更时才会调用纯管道的 `transform()` 方法来实现数据转换，从而将数据更新到页面上。纯变更是指对基本数据类型（String、Number、Boolean 等）输入值的变更或对对象引用（Date、Array、Function、Object 等）的更改。

现以 `DatePipe` 日期管道为例，分别用 String 类型和 Date 类型的对象作为输入值，对日期进行格式化。同时设定一个 2 秒的定时器，用来动态改变日期的月份。示例代码如下：

```
// ...
```

```
@Component({
  selector: 'pure-pipe-demo',
  template: `
    <div>
      <p>{{dateObj | date:"y-MM-dd HH:mm:ss EEEE"}}</p>
      <p>{{dateStr | date:"y-MM-dd HH:mm:ss EEEE"}}</p>
    </div>
  `
})
export class PurePipeDemoComponent {
  dateObj: date = new Date('2016-06-08 20:05:08');
```

```
dateStr: string = '2016-06-08 20:05:08';
constructor() {

  setTimeout(() => {
    this.dateObj.setMonth(11);
    this.dateStr = '2016-12-08 20:05:08';
  }, 2000);
}
```

在上面的例子中，日期管道输入值的初始日期都为“2016-06-08 20:05:08”，最初页面效果如下：

```
'2016-06-08 20:05:08 Wednesday'
'2016-06-08 20:05:08 Wednesday'
```

待 2 秒后页面显示为：

```
'2016-06-08 20:05:08 Wednesday'
'2016-12-08 20:05:08 Thursday'
```

当 2 秒计时完成后，模板中的日期输入值都被修改成 2016-12-08 20:05:08，但显示在页面上的数据却是不同的。因为 String 对象（dateStr）的引用发生了变化，被赋值为另一个常量字符串，而 Date 对象（dateObj）的引用没有变化。在模板表达式中使用纯管道 DatePipe，只有当输入值发生纯变更后才会调用该管道并更新变化的值。

纯管道的变化监测策略是基于判断基本类型的数据值或对象的引用是否被改变来监测对象变化的。对象引用的监测方式比遍历对象内部所有属性值的监测方式要快得多，Angular 使用的是对象引用的监测策略，这样能快速地判断是否可以跳过执行管道并更新视图。

非纯管道

使用非纯管道（Impure Pipe），Angular 组件在每个变化监测周期都会调用非纯管道，并执行管道的 transform() 方法来更新页面数据。可以在管道元数据里将 pure 属性值设置为 false 来定义非纯管道。示例代码如下：

```
@Pipe({
  name: 'selectContact',
  pure: false
})
```

在上述过滤联系人的例子中，给管道添加 `pure: false` 就可以将其定义为非纯管道。非纯管道在每个变化周期内都会去监测并执行 `selectContact` 管道的 `transform()` 方法，对发生变化的联系人列表数据进行过滤，最终将联系人列表数据同步到模板视图中。

在 Angular 的内置管道中，`SlicePipe`、`AsyncPipe` 和 `JsonPipe` 属于非纯管道，下面通过一个非纯异步管道（`AsyncPipe`）例子来理解非纯管道。非纯异步管道需要接收 `Promise` 或 `Observable` 对象作为输入，并自动订阅这个输入，最终返回该异步操作产生的值。示例代码如下：

```
import { Component, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs/Rx';

@Component({
  selector: 'impure-pipe-demo',
  template: `
    <p> 时间: {{ time | async }}</p>
  `
})
export class ImpurePipeDemoComponent implements OnInit {
  time: Observable<string>;
  constructor() {}
  ngOnInit() {
    this.time = new Observable<string>((observer: Subscriber<string>) => {
      setInterval(() => observer.next(new Date().toLocaleString()), 1000);
    });
  };
}
```

在这个例子中使用非纯异步管道把一个时间字符串 `time` 的 `Observable` 绑定到视图中，通过异步管道实现了每隔 1 秒切换一次时间的时钟效果。



与 `Observable` 相关的内容会在第 9 章中详细讲解，这里不再展开，读者可以在理解了 `Observable` 及 `RxJS` 的相关内容后再回来理解这个示例。

7.6 扩展阅读

7.6.1 安全导航操作符

Angular 的模板表达式在某些特定场景中允许使用一些特殊的连接操作符，如本章介绍的管道操作符。接下来将学习另一种特殊的连接操作符，即安全导航操作符“?”。

假设模板表达式 `detail.telNum` 的值为 `0123456789`，下面的代码正常运行后界面将会显示为 `0123456789`：

```
<p>{{ detail.telNum }}</p>
```

如果模板变量 `detail` 没被赋值，那么在 Angular 中会因为报错而导致程序无法运行，从而使页面渲染失败。实际上，我们可以通过下面的处理方式来规避这种错误。示例代码如下：

```
<p>{{ detail && detail.telNum }}</p>
```

这种处理方式在语法上没有问题，结果也能达到预期。但是当碰到像 `a.b.c.d` 这种长属性路径时，继续使用这种方式将会导致代码较为臃肿，后期维护成本也会倍增。Angular 的安全导航操作符“?”可以用来规避因为属性路径中出现 `null` 或者 `undefined` 值而导致的错误，比如之前的代码可以用更优雅的方式书写如下：

```
<p>{{ detail?.telNum }}</p>
```

7.6.2 双向绑定的原理

在本章的前面内容中我们已经了解到，`[(ngModel)]` 可以拆分为 `ngModel` 和 `ngModelChange` 两部分。其中，`ngModel` 作为 `NgModel` 指令的输入属性用来设置元素的值；`ngModelChange` 作为 `NgModel` 指令的输出属性用来监听元素值是否变化。

需要注意的是，`ngModelChange` 属性并不会生成 DOM 事件，实际上它是一个 `EventEmitter` 类型的对象。`[(ngModel)]` 的具体实现如下：

```
@Directive({
  selector: "[ngModel]",
  host: {
    "[value]": "ngModel",
    "(input)": "ngModelChange.next($event.target.value)"
  }
})
```

```
class NgModelDirective{
  @Input() ngModel:any;
  @Output() ngModelChange:EventEmitter = new EventEmitter();
}
```

上面的例子涉及第 8 章指令的相关知识，其中 `host` 属性用来描述和指令元素相关的输入、输出属性变化，即当 `[ngModel]` 的 `ngModelChange` 事件发生时就会触发 `input` 事件，当 `[ngModel]` 的 `ngModel` 值变化时就会更新 `value` 属性。

Angular 提供了一种双向数据绑定的语法，即 `[(x)]`。也就是说，当 Angular 解析一个 `[(x)]` 的绑定目标时，相当于为这个 `x` 指令绑定一个名为 `x` 的输入属性和一个名为 `xChange` 的输出属性。示例代码如下：

```
<span [(x)]="e"></span>
```

// 等同于下面的代码

```
<span [x]="e" (xChange)="e=$event"></span>
```

总的来说，双向数据绑定实际上就是通过输入属性存储数据，同时通过一个与之对应的输出属性（输入属性 + `Change` 后缀）监听输入属性的数据变化来触发相应的事件。

了解了 Angular 的双向数据绑定原理后，接下来通过创建一个支持双向绑定的组件例子来加深理解。该例子绑定一个 `number` 的输入、输出属性，同时在组件中需要定义一个 `@Output` 输出属性来匹配 `@Input` 输入属性。示例代码如下：

```
// amount.component.ts
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'amount',
  template: `
    <span>
      子组件当前值: {{value}} -
      <button (click)="increment()"> 增加 </button>
    </span>
  `,
})
export class AmountComponent {
  @Input() value: number = 0;
  @Output() valueChange: EventEmitter<number> = new EventEmitter<number>();
```

```
    increment() {
      this.value++;
      this.valueChange.emit(this.value);
    }
  }

// app.component.ts
import { Component, Input } from '@angular/core';
import { AmountComponent } from './amount.component';

@Component({
  selector: 'app',
  template: `
    <div>
      <div>
        <span>Number 1:</span>
        <amount [(value)]="number1"></amount>
      </div>
      <div>
        <span>Number 2:</span>
        <amount [value]="number2" (valueChange)="number2=$event"></amount>
      </div>
      <ul>
        <li>Number 1: 父组件当前值: {{number1}} </li>
        <li>Number 2: 父组件当前值: {{number2}} </li>
      </ul>
    </div>
  `,
})
export class Parent {
  number1: number = 0;
  number2: number = 1;
}
```

7.7 小结

本章主要关注模板特性及模板周边的一些功能用法。首先从常见的模板语法讲起，一一罗列了模板的基本语法特性，如数据绑定、模板表达式及模板语句等。然后详细讲述了 5 个常用的内置指令（NgClass、NgStyle、NgIf、NgSwitch、NgFor）的用法。接下

来介绍了表单，表单有两种类型：模板驱动和模型驱动，本章重点讲述了模板驱动型表单，涵盖了表单指令、表单状态跟踪、表单校验及表单双向绑定等多种应用场景。最后介绍了模板的另一个重要特性：管道，讲述了管道的基本用法、常见的内置管道命令，以及自定义管道的编写，并介绍了纯管道和非纯管道在性能优化方面的应用。

指令 8

在 Angular 中，指令是一个重要的概念，它作用在特定的 DOM 元素上，可以扩展这个元素的功能，为元素增加新的行为。本书第 6 章介绍过组件的知识。本质上，组件可以被理解作为一种带有视图的指令。组件继承自指令，是指令的一个子类，通常被用来构造 UI 控件。

本章内容聚焦于指令，将详细介绍 Angular 中指令的分类、内置指令的使用及自定义指令等相关内容。

8.1 概述

作为 Web 开发者，对 HTML 都有基本的认识，接下来将对 HTML 相关内容做一个简单的回顾。

HTML 文档

HTML 文档是一个纯文本文件，包含了 HTML 元素、CSS 样式及 JavaScript 代码。

HTML 元素

HTML 文档是由 HTML 元素定义的。HTML 元素指的是从开始标签到结束标签的所有代码，元素的内容是开始标签与结束标签之间的内容，如表 8-1 所示。

表 8-1 HTML 标签

开始标签	元素内容	结束标签
<code></code>	This is a link	<code></code>

HTML 属性

HTML 标签可以设置属性，属性为 HTML 元素提供了更多附加信息。属性一般以名称="值"的形式出现，比如[href="http://www.google.com"](http://www.google.com)。有时也会只有名称而没有值，比如 disabled。以超链接元素为例，HTML 超链接由 `<a>` 标签定义，链接的地址在 href 属性中指定。示例代码如下：

```
<a href="http://www.google.com">单击此处打开 Google</a>
```

超链接标签创建了一个从一个页面到另一个页面（或者本页面）的链接，href 作为超链接标签的属性，定义了链接的目标地址。以上述代码为例，当用户单击这个超链接时，浏览器地址栏的 URL 被修改为 www.google.com 并加载 Google 的首页。

上述行为的发生，依赖浏览器在解析 HTML 文档时，可以正确理解 HTML 标签元素 `<a>` 的内容。浏览器遵循 HTML 标准，理解 `<a>` 标签声明了一个超链接，href 属性值指定了链接的目标地址。

下面实现一个自定义指令 `BeautifulBackgroundDirective`，它可以为元素添加一个好看的背景色。示例代码如下：

```
<a href="http://www.google.com" myBeautifulBackground>单击此处打开 Google</a>
```

上述示例为 `<a>` 标签增加了一个属性 `myBeautifulBackground`，通过这样的方式使用定义好的 `BeautifulBackgroundDirective` 指令。在这里读者先不用深入探究 `BeautifulBackgroundDirective` 指令的完整实现，仅需了解它的使用方式即可。

通过上述内容可以看出，指令的使用并不复杂，它与 HTML 元素属性的使用方式相似。不同的是，HTML 语法标准为 HTML 元素预定义了特定的属性，浏览器遵循这一语法标准，实现了这些属性的内置行为。语法标准预定义的属性是有限的、不可扩展的，而 Angular 中的指令是可自定义的、可任意扩展的，这在一定程度上弥补了标准 HTML 元素属性功能的不足。本章接下来的内容将会深入讲解如何使用指令来扩展 HTML 元素的功能。

8.1.1 指令分类

在 Angular 中包含三种类型的指令，分别是属性指令、结构指令和组件。

属性指令

顾名思义，属性指令是以元素属性的形式来使用的指令。与 HTML 元素的内置属性不同，指令是 Angular 对 HTML 元素属性的扩展，浏览器本身不能识别这些指令，指令仅在 Angular 环境中才能被识别使用。属性指令通常被用来改变元素的外观和行为，如在第 7 章中介绍过的 Angular 内置指令 `NgStyle`，它可以基于组件的状态来动态设置目标元素的样式。

结构指令

结构指令可以用来改变 DOM 树的结构。结构指令可以根据模板表达式的值，增加或删除 DOM 元素，从而改变 DOM 的布局。结构指令与属性指令的使用方式相同，都是以元素属性的形式来使用的。两者的区别在于使用场景不同，属性指令用来改变元素的外观和行为，而结构指令用来改变 DOM 树的结构。

以 Angular 内置的结构指令 `NgIf` 为例，使用 `NgIf` 指令需要为指令绑定一个表达式，当表达式值为 `true` 时，该 DOM 元素及其子元素被添加至 DOM 中；当表达式值为 `false` 时，元素从 DOM 中被移除。示例代码如下：

```
<!-- 示例 A -->
<p *ngIf="condition">
  condition 的值为 true，读者能看到这句话。
</p>

<!-- 示例 B -->
<p *ngIf="!condition">
  !condition 的值为 false，读者不能看到这句话。
</p>
```

在上述代码示例中，当 `condition` 为 `true` 时，示例 A 中 `ngIf` 绑定的表达式结果为 `true`，示例 B 中表达式结果为 `false`，因此，仅示例 A 中的段落元素被添加至 DOM 中。

组件

在第 6 章中详细介绍了与组件相关的内容，组件是被用来构造带有自定义行为的可重用视图。组件与指令的结构类似，均使用装饰器描述元数据，二者均在各自对应的类中实现具体的业务逻辑。先来看看组件和指令的基本结构。示例代码如下：

```
// Component
@Component({
  selector: 'hello-world',
  template: '<div>Hello world</div>'
})

class HelloWorldComponent {
  // ...
}

// Directive
@Directive({
  selector: 'myHelloWorld'
})

class HelloWorldDirective {
  // ...
}
```

可以看出，组件与指令的基本结构非常相似，差别在于组件中包含了模板。组件作为指令的一个子类，它的部分生命周期钩子与指令的相同，相关的钩子方法说明如表 8-2 所示。

表 8-2 组件与指令相同的生命周期钩子方法

钩子方法	作用
ngOnInit	在 Angular 完成初始化输入属性的数据绑定后，初始化指令/组件
ngOnChanges	在 Angular 初始化输入属性的数据绑定前响应一次，之后当检测到数据绑定发生变化时就会被调用，这个方法接收一个包含当前和之前数据的 SimpleChanges 对象
ngDoCheck	用于变化监测，该钩子方法会在每次变化监测发生时被调用
ngOnDestroy	在 Angular 销毁指令/组件之前执行清理工作，此时应注销观察者对象或者解绑事件处理器以避免内存泄漏

尽管组件与指令有相似的结构和一些相同的生命周期钩子方法，但是它们也有一些不同点。不同于属性指令和结构指令，组件不是以 HTML 元素属性的形式使用的，而是以自定义标签的形式使用的，原因在于组件带有模板。组件可作为对 HTML 元素的扩展，将自身的模板视图插入 DOM 中；而属性指令和结构指令是对 HTML 元素属性的扩展，其作用是扩展已有 DOM 元素的行为和样式，或者改变这些元素在 DOM 中的结构。

正是因为指令中不具有模板，因此在组件中，如下围绕模板视图的初始化和更新的生命周期钩子方法是组件独有而指令所不具有的。

- `ngAfterContentInit`
- `ngAfterContentChecked`
- `ngAfterViewInit`
- `ngAfterViewChecked`

8.1.2 内置指令

为了帮助用户快速编写应用，Angular 内置了一些常用的指令。根据这些指令使用场景的不同，可将其划分为下列三个类别：

- 通用指令
- 路由指令
- 表单指令

各类别包含的内置指令如图 8-1 所示。

本节将依次对各个类别及其包含的内置指令进行讲解。

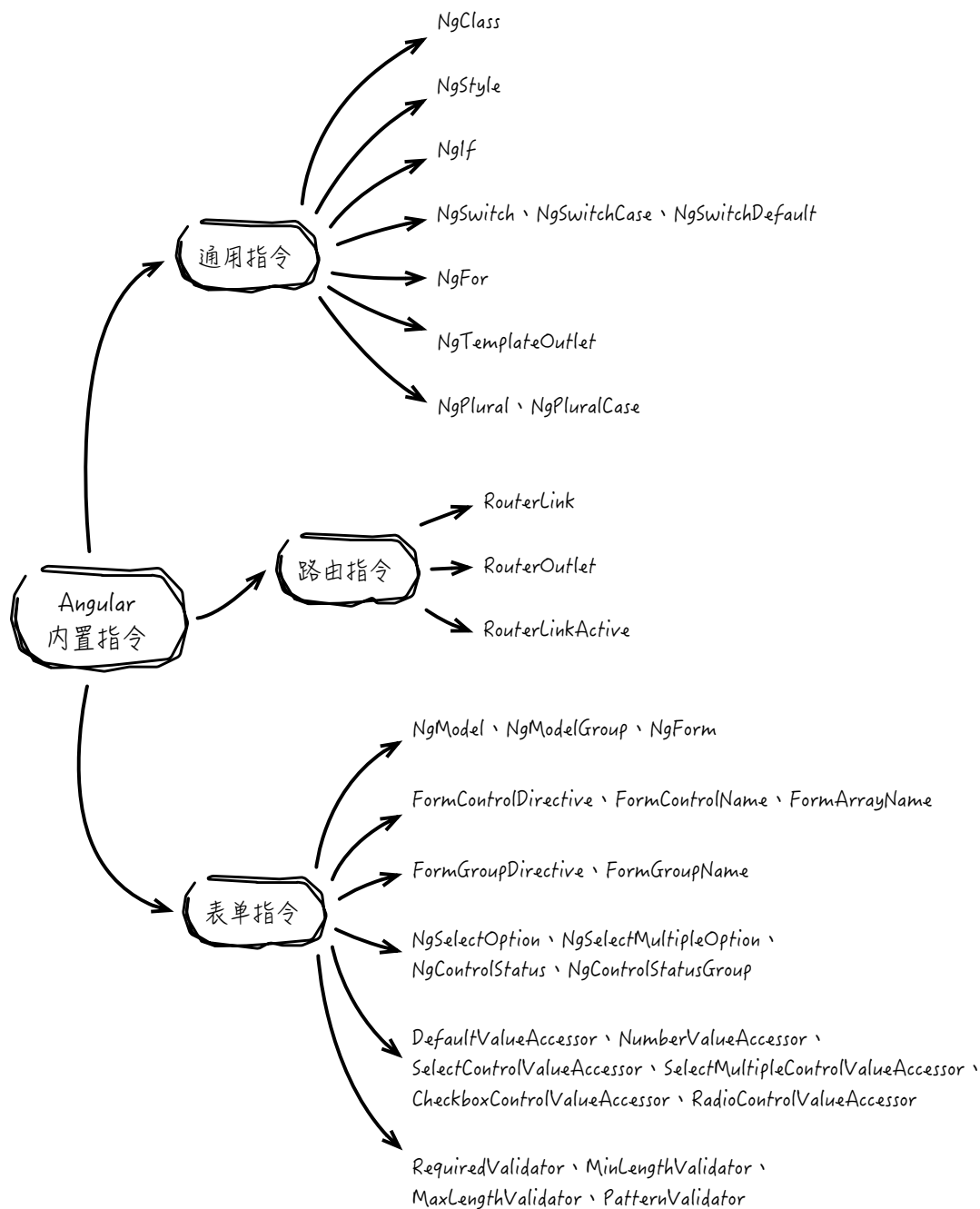


图 8-1 Angular 的内置指令

通用指令

通用指令是指在 Angular 应用中经常会用到的指令，它包含的内置指令如图 8-2 所示。

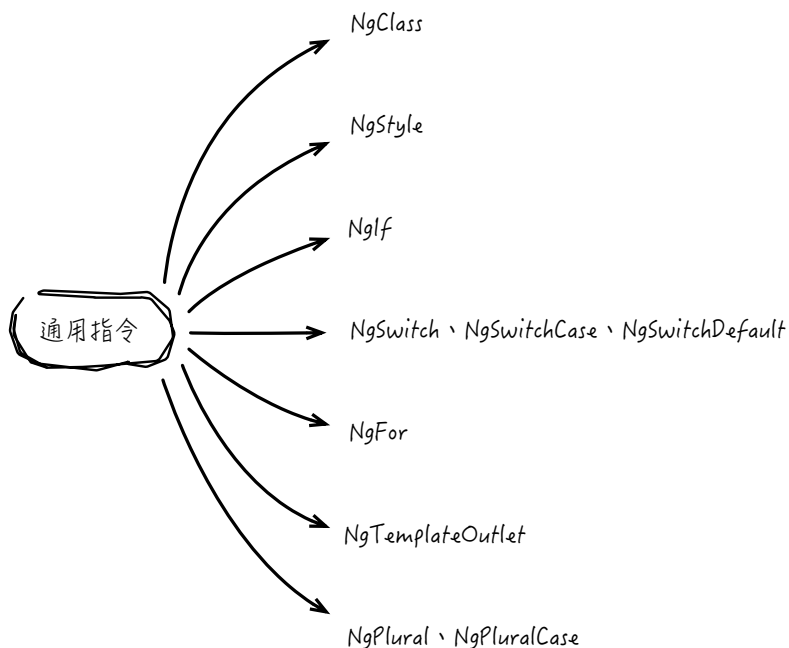


图 8-2 Angular 的通用指令

Angular 将上述通用指令包含在 CommonModule 模块中，当需要使用通用指令时，首先应当在应用的模块中引入 CommonModule 模块。例如，当需要使用 NgIf 指令时，引用方式如下：

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
```

```
export class AppModule {}

// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <p *ngIf="condition">当 condition 的值为 true, 读者能看到这句话。</p>
  `
})
export class AppComponent {
  condition = true;
}
```

在上述 `app.module.ts` 的 `@NgModule` 装饰器中, `imports` 属性没有主动引入 `CommonModule` 模块。这是因为 Angular 的 `BrowserModule` 模块已包含了 `CommonModule` 模块, 通过引入 `BrowserModule` 模块, 已经间接引入了 `CommonModule`, 因此在 `AppComponent` 组件中, 可以直接使用 `NgIf` 指令。

通用指令包含了各种常用指令, 其中 `NgClass`、`NgStyle`、`NgIf`、`NgSwitch`、`NgFor` 指令已在“模板”章节的内置指令部分详细讲解过, 这里不再赘述。除此之外, `NgTemplateOutlet` 指令和 `NgPlural`、`NgPluralCase` 指令属于相对较生僻的指令, 应用场景相对较少, 在本章“扩展阅读”部分会对它们进行讲解。

路由指令

路由指令包含 Angular 路由中需要用到的指令, 如图 8-3 所示。

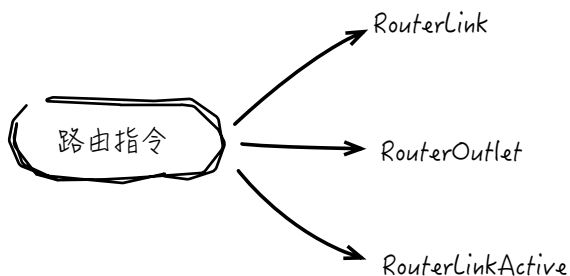


图 8-3 Angular 的路由指令

图 8-3 中的 RouterOutlet 指令是一个占位符，当路由跳转时，Angular 会查找当前匹配的组件并将组件插入 RouterOutlet 中；RouterLinkActive 指令在当前路径与元素设置的链接匹配时为元素添加 CSS 样式；RouterLink 指令使得应用可以链接到特定组件。

在本书的“路由”章节，会对上述路由指令的用法进行具体讲解，这里不再赘述。

表单指令

表单指令包含了一系列可以在 Angular 表单中使用的指令，这些指令分别被包含在下列三个模块中：

- FormsModule
- ReactiveFormsModule
- InternalFormsSharedModule

FormsModule 模块

FormsModule 模块包含了 NgModel、NgModelGroup、NgForm 指令和 InternalFormsSharedModule 模块包含的指令，如图 8-4 所示。

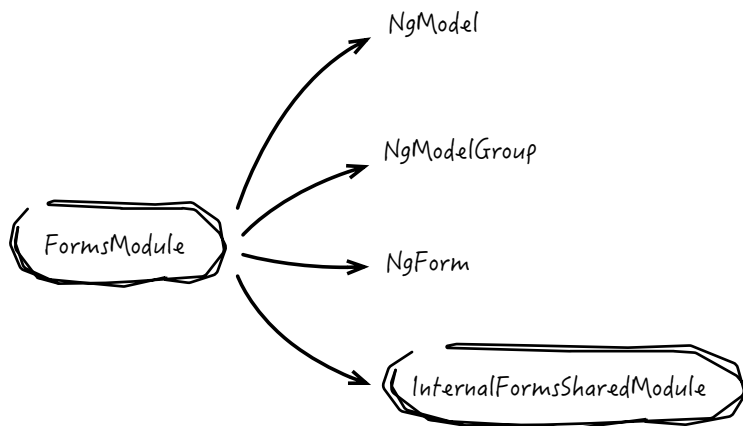


图 8-4 FormsModule 模块包含的指令

在第 7 章的“表单”部分已经对 NgForm、NgModel、NgModelGroup 指令进行了详细讲解，这里不再赘述。

ReactiveFormsModule 模块

ReactiveFormsModule 模块包含了 FormControlDirective、FormGroupDirective、FormControlName、FormGroupName、FormArrayName 指令和 InternalFormsSharedModule 模块包含的指令，如图 8-5 所示。

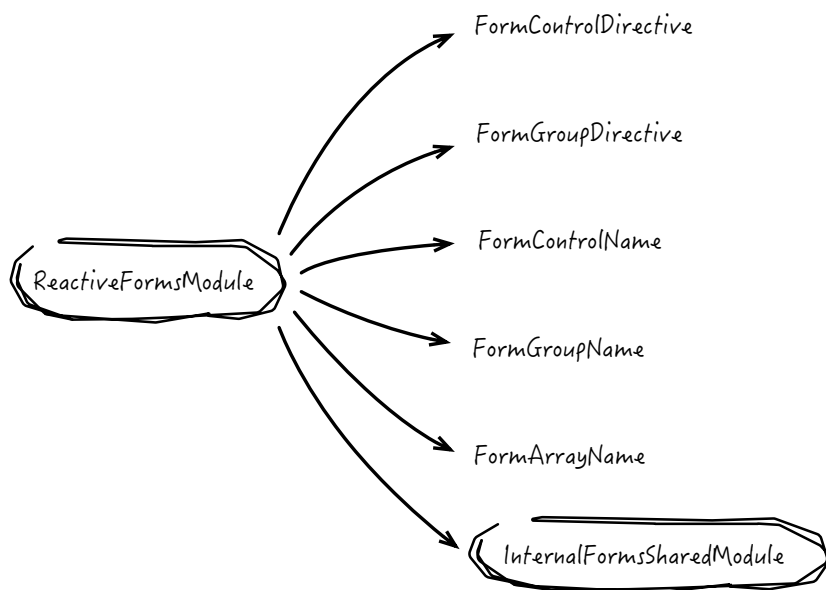


图 8-5 ReactiveFormsModule 模块包含的指令

FormControlDirective 指令

FormControlDirective 指令可以将一个已有的 FormControl 实例绑定到一个 DOM 元素。

在下面示例中，将声明的 loginControl 绑定给了一个输入元素，当输入元素的值变化时，loginControl 的值进行相应的变化；同时，当 loginControl 控件值变化时，输入元素的值也发生相应的变化。示例代码如下：

```
@Component({
  selector: 'my-app',
  template: `
    <div>
      <h2>FormControl 例子</h2>
      <form>
```

```

        <p>绑定到 input 标签: <input type="text" [formControl]="loginControl"></p>
        <p>获得 input 的值: {{loginControl.value}}</p>
    </form>
</div>
、
})
export class App {
    loginControl: FormControl = new FormControl('');
}

```

FormGroupDirective 指令

FormGroupDirective 指令可以将一个已有的表单组合绑定到一个 DOM 元素。

以下示例中，将声明的 loginForm 绑定到了表单元素，并且将 loginForm 表单组合的用户名和密码输入控件分别绑定给了 name 和 password 输入元素。示例代码如下：

```

@Component({
    selector: 'my-app',
    template: `
        <div>
            <h2>FormGroup 例子</h2>
            <form [formGroup]="loginForm">
                <p>用户名: <input type="text" formControlName="name"></p>
                <p>密码: <input type="password" formControlName="password"></p>
            </form>
            <p>LoginForm 的值: </p>
            <pre>{{ loginForm.value | json}}</pre>
        </div>
    `
})
export class App {
    loginForm: FormGroup;
    constructor() {
        this.loginForm = new FormGroup({
            name: new FormControl(""),
            password: new FormControl("")
        });
    }
}

```

FormControlName 指令

FormControlName 指令将一个已有的表单控件与一个 DOM 元素绑定，在绑定时使用 FormControlName 指令为表单控件指定一个别名。这一指令仅作为 FormGroupDirective 指令的子元素使用，在上面对 FormGroupDirective 指令的讲解中，将 loginForm 表单组合的用户名和密码输入控件分别绑定给了 name 和 password 输入元素，并使用 FormControlName 指令分别指定了别名。

除此之外，FormGroupName 指令和 FormArrayName 指令属于相对生僻的指令，在本章“扩展阅读”中会对其进行讲解。

InternalFormsSharedModule 模块

InternalFormsSharedModule 模块是 Angular 的内部模块，FormsModule 和 ReactiveFormsModule 模块均引入了 InternalFormsSharedModule 模块，因此当引入了 FormsModule 和 ReactiveFormsModule 模块时，可以使用 InternalFormsSharedModule 模块包含的指令。InternalFormsSharedModule 模块包含的指令如图 8-6 所示。

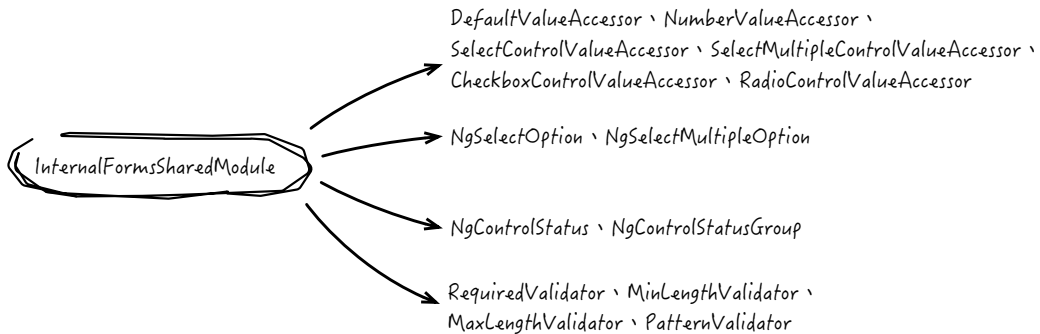


图 8-6 InternalFormsSharedModule 模块中的指令

在 InternalFormsSharedModule 模块包含的指令中，第一部分为表单元素访问器指令，它包含的指令如下：

- DefaultValueAccessor
- NumberValueAccessor
- CheckboxControlValueAccessor
- RadioControlValueAccessor
- SelectControlValueAccessor、SelectMultipleControlValueAccessor

这些访问器指令是 Angular 表单的内部指令，在应用中无须主动使用它们，它们是 DOM 元素和表单输入控件之间的桥梁。其中 `ControlValueAccessor` 是其他访问器指令的父接口，抽象了 Angular 控件与 DOM 元素交互的公共方法。`ControlValueAccessor` 接口定义了三个方法，即向 DOM 元素写入新值的方法 `writeValue()`、用于监听 DOM 元素值变更的方法 `registerOnChange()`，以及用于监听 DOM 元素触摸事件的方法 `registerOnTouched()`。这三个方法在接口中定义如下：

```
export interface ControlValueAccessor {  
  writeValue(obj: any) : void;  
  registerOnChange(fn: any) : void;  
  registerOnTouched(fn: any) : void;  
}
```

对应不同的数据输入类型，表单指令集合提供了各自的访问器类，如下所示。

- `DefaultValueAccessor` 为 text 文本输入框的访问器。
- `CheckboxControlValueAccessor` 为 checkbox 复选框的访问器。
- `RadioControlValueAccessor` 为 radio 单选钮的访问器，`RadioButtonState` 保存单选钮的选中状态和选中的值。
- `NumberValueAccessor` 为 number 数字输入框的访问器。
- `SelectControlValueAccessor` 为 select 下拉框的访问器，`NgSelectOption` 指令动态地标记 `<option>` 选项，当选项变更时，Angular 会收到通知。

在 `InternalFormsSharedModule` 模块包含的指令中，第二部分为选择框选项指令，它包含的指令如下：

- `NgSelectOption`
- `NgSelectMultipleOption`

上述指令为 Angular 的内部指令，它们动态地标记选择框的 `<option>` 选项，当选项变更时，Angular 会收到通知。有了这些内部指令的帮助，在表单的选择框中可以设置多选框和单选框。

在 `InternalFormsSharedModule` 模块包含的指令中，第三部分为表单验证指令，它包含的指令如下：

- `RequiredValidator` 指令，为表单输入元素增加 `required` 约束。
- `MinLengthValidator` 指令，为表单输入元素增加 `minlength` 约束。
- `MaxLengthValidator` 指令，为表单输入元素增加 `maxlength` 约束。

- PatternValidator 指令，为表单元素增加正则表达式约束，输入内容必须符合正则表达式定义的模式。

在 InternalFormsSharedModule 模块包含的指令中，最后一部分为控件状态指令，它包含的指令如下：

- NgControlStatus
- NgControlStatusGroup

上述指令是 Angular 表单的内部指令，在应用中无须主动使用它们，它们会自动根据控件是否通过验证、是否被触摸等状态来设置元素的 CSS 类。

8.2 自定义属性指令

Angular 提供了不少功能强大的内置指令，但在现实情况中，这些内置指令可能还不能完全满足要求，这时就需要我们编写自定义指令来实现特定的需求。本节内容将实现一个自定义属性指令，通过使用这一指令，可以实现在单击按钮时改变其背景色的功能。

8.2.1 实现属性指令

一个属性指令需要一个控制器类，该控制器类使用 @Directive 装饰器来装饰。@Directive 装饰器指定了用以标记指令所关联属性的选择器，控制器类则实现了指令所对应的特定行为。

新建指令文件 beautifulBackground.directive.js，示例代码如下：

```
// beautifulBackground.directive.js
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[myBeautifulBackground]'
})
export class BeautifulBackgroundDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

在上述代码示例中，首先从 Angular 核心模块 `@angular/core` 中引入了 `Directive` 和 `ElementRef`。`Directive` 被用作 `@Directive` 装饰器，而 `ElementRef` 则用来访问 DOM 元素。随后，在 `@Directive` 装饰器中以键值对的形式定义了指令的元数据。在配置对象中，使用 `selector` 属性来标识该属性指令所关联的元素名称，`[myBeautifulBackground]` 是指令所对应的 CSS 选择器，注意中括号 “[]” 不能丢，中括号在 CSS 选择器中表示元素属性匹配。所以当指令运行时，Angular 会在模板中匹配所有包含属性名称 `myBeautifulBackground` 的 DOM 元素。

在 `@Directive` 元数据下面是该自定义指令的控制器类，该类实现了指令所包含的逻辑。`export` 关键字被用来将指令导出以供其他组件访问。

Angular 会为每一个匹配的 DOM 元素创建一个指令实例，同时将 `ElementRef` 作为参数注入到控制器构造函数。使用 `ElementRef` 服务，可以在代码中通过其 `nativeElement` 属性直接访问 DOM 元素，这样就可以通过 DOM API 设置元素的背景色了。为简单起见，这里仅为元素设置了一种背景色，在实际项目代码中，可以根据需要设置更为复杂的样式效果。

现在已经实现了基础的自定义指令，接下来可以把它应用到组件中。示例代码如下：

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: '<div myBeautifulBackground>这是一个按钮</div>'
})
export class AppComponent { }

// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { BeautifulBackgroundDirective } from './beautifulBackground.directive';

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, BeautifulBackgroundDirective],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

在上述代码中，首先在 `AppModule` 中引入自定义指令代码文件，并在 `@NgModule` 装饰器中为 `declarations` 属性赋值声明引用。然后在 `AppComponent` 组件的模板中，在将要使用指令的目标元素上添加 `myBeautifulBackground` 属性，这个属性名需要与自定义指令的 `@Directive` 装饰器中选择器内容匹配。

运行这个示例，Angular 会在组件的 `<div>` 元素上解析到 `myBeautifulBackground` 属性，然后创建 `BeautifulBackgroundDirective` 指令类的实例，将元素的引用传入构造函数，用以设置元素的样式。

8.2.2 为指令绑定输入

在上述自定义指令中，为元素设置的背景色是固定的。为了增加灵活性，可以在指令外部使用绑定的方式设置背景色。示例代码如下：

```
<div [myBeautifulBackground]="color">这是一个按钮</div>
```

为了给指令绑定外部变量，需要为指令声明一个可绑定的输入属性 `backgroundColor`。需要在属性上使用 `@Input` 装饰器，`@Input` 标识使得属性具有绑定能力，可将外部变量的值绑定到指令的属性中。

```
// ...
```

```
@Directive({
  selector: '[myBeautifulBackground]'
})
export class BeautifulBackgroundDirective {
  @Input('myBeautifulBackground')
  backgroundColor: string;
  // ...
}
```

在上述代码的 `@Input('myBeautifulBackground')` 装饰器中，为输入属性 `backgroundColor` 指定了 `myBeautifulBackground` 别名。注意这里指定的别名，与指令在 `@Directive` 装饰器中定义的选择器名称一致。这并不是必需的。这里定义成一致的，是为了在元素中使用指令时，不必再使用另外的名称来绑定输入变量了。当然，也可以为 `backgroundColor` 属性定义其他的别名，如将属性别名定义为 `@Input('myBackgroundColor')`，这时需要在组件中使用如下方式来绑定这一属性：

```
<div myBeautifulBackground [myBackgroundColor]="color">这是一个按钮</div>
```

在上述示例中，为绑定属性名和别名分别命名了不同的名称。如果不需要为属性提供不同的别名，则可以在 `@Input` 装饰器中省略对别名的定义，使用如下所示的便捷定义：

```
// ...
@Input()
backgroundColor: string;
// ...
```

那么上述省略了对别名的定义后，在模板中用法如下：

```
<div myBeautifulBackground [backgroundColor]="color">这是一个按钮</div>
```

现在已经为指令定义了用来绑定输入变量的属性，在组件模板中将 `color` 变量绑定给这一属性。`color` 变量可以来自多个地方，比如可以是在组件中定义的变量，也可以是来自组件模板的输入元素。如单击某一单选按钮时，将单选按钮的结果赋值给 `color` 变量。示例代码如下：

```
<div>
  <input type="radio" name="colors" (click)="color='green'">绿色
  <input type="radio" name="colors" (click)="color='yellow'">黄色
  <input type="radio" name="colors" (click)="color='red'">红色
</div>
<div [myBeautifulBackground]="color">这是一个按钮</div>
```

同时，为了在输入属性 `backgroundColor` 的值变更时相应地改变元素样式，可以重写 `backgroundColor` 属性的 `set` 方法。示例代码如下：

```
import { Directive, ElementRef, Input } from '@angular/core';

@Directive({
  selector: '[myBeautifulBackground]'
})

export class BeautifulBackgroundDirective {
  private _defaultColor = 'yellow';
  private el: HTMLElement;

  // 重写了 backgroundColor 属性的 set 方法
  @Input('myBeautifulBackground') set backgroundColor(colorName: string) {
    this.setStyle(colorName);
  }
}
```



```
constructor(el: ElementRef) {  
    this.el = el.nativeElement;  
    this.setStyle(_defaultColor);  
}  
  
private setStyle(color: string) {  
    this.el.style.backgroundColor = color;  
}  
}
```

8.2.3 响应用户操作

在为指令绑定了输入属性后，可以根据输入值动态地变更元素的样式。本节将继续丰富指令的功能，使得自定义指令可以响应用户的操作。接下来将实现用户单击按钮时改变元素样式的例子，为了实现这种效果，需要在事件处理函数上添加 `@HostListener` 装饰器。

```
// ...  
@HostListener('click')  
onClick() {  
    // ...  
}  
// ...
```

`@HostListener` 装饰器指向使用属性指令的 DOM 元素，使得 DOM 元素的事件与指令关联起来。除此之外，也可以直接为 DOM 元素添加事件监听器，但不建议这样做，因为这样做至少存在以下三个问题：

- 必须正确地编写监听器。
- 必须在指令销毁前移除监听器，避免内存泄漏。
- 会直接操作 DOM API，这是需要避免的方式。

因此，更合理的方式是使用 `@HostListener` 装饰器来实现这个事件处理函数。示例代码如下：

```
// ...  
@HostListener('click')  
onClick() {  
    this.setStyle(this.backgroundColor || this._defaultColor);  
}  
// ...
```

现在可以根据用户操作来改变元素的样式了，而不是在输入属性变更时触发样式更新。修改过的指令示例代码如下：

```
import { Directive, ElementRef, Input } from '@angular/core';

@Directive({
  selector: '[myBeautifulBackground]'
})

export class BeautifulBackgroundDirective {
  private _defaultColor = 'yellow';
  private el: HTMLElement;

  @Input('myBeautifulBackground') backgroundColor: string;

  constructor(ref: ElementRef) {
    this.el = ref.nativeElement;
    this.setStyle(this._defaultColor);
  }

  @HostListener('click') onClick() {
    this.setStyle(this.backgroundColor || this._defaultColor);
  }

  private setStyle(color: string) {
    this.el.style.backgroundColor = color;
  }
}
```

使用上述指令，当用户单击元素时，可以根据绑定的输入属性的值更新元素的样式。如以下模板代码，在初始时这个 div 的背景色是黄色（yellow），当用户单击该 div 时，其颜色就会变成红色（red）：

```
<div [myBeautifulBackground]="red">这是一个按钮</div>
```

8.3 自定义结构指令

本章 8.2 节中实现了一个自定义属性指令，使用它可以在单击元素时改变其样式。除了可以自定义属性指令，Angular 还允许自定义结构指令，本节将介绍自定义结构指

令的相关内容，并实现一个自定义结构指令。作为示例，接下来将实现一个和 `NgIf` 指令作用相反的指令。

8.3.1 实现结构指令

本节将要实现一个自定义结构指令：`Unless` 指令。当表达式的值为 `false` 时，渲染模板内容；当值为 `true` 时，将模板内容从 DOM 树中移除。

与 8.2 节中的自定义属性指令类似，创建自定义结构指令涉及以下内容：

- 引入 `@Directive` 装饰器。
- 添加 CSS 属性选择器，用来标识指令。
- 声明一个 `input` 属性，用来绑定表达式。
- 将装饰器应用在指令实现类上。

按照上述内容，初始代码如下：

```
import { Directive, Input } from '@angular/core';

@Directive({selector: '[myUnless]'})
export class UnlessDirective {
  @Input('myUnless') condition: boolean;
}
```

`Unless` 指令需要访问组件模板内容，并且需要可以渲染组件模板内容的工具。使用 `TemplateRef` 和 `ViewContainerRef` 服务可以实现这一目标，其中 `TemplateRef` 可以用来访问组件模板，而 `ViewContainerRef` 可作为视图内容渲染器，将模板内容插入 DOM 中。`TemplateRef` 和 `ViewContainerRef` 服务来自 `@angular/core`，在指令的构造函数中需要将它们作为依赖注入，赋值给指令的变量。示例代码如下：

```
// ...

constructor(
  private templateRef: TemplateRef<any>,
  private viewContainer: ViewContainerRef
){}

// ...
```

在组件中使用 `Unless` 指令时，需要为指令绑定一个值为布尔类型的表达式或变量，指令根据绑定结果增加或者删除模板内容。为了在接收到绑定结果时实现这一逻辑，需

要为 `condition` 属性设置一个 `set` 方法。示例代码如下：

```
@Input('myUnless')
set condition(newCondition: boolean) {
  if (!newCondition) {
    this.viewContainer.createEmbeddedView(this.templateRef);
  } else {
    this.viewContainer.clear();
  }
}
```

在上述代码中，通过调用渲染器的 `createEmbeddedView()` 和 `clear()` 方法，实现了根据输入属性的值添加和删除模板内容的功能。

最终 `Unless` 指令的示例代码如下：

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[myUnless]'
})
export class UnlessDirective {
  @Input('myUnless')
  set condition(newCondition: boolean) {
    if (!newCondition) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }

  constructor(
    private templateRef: TemplateRef < any > ,
    private viewContainer: ViewContainerRef
  ) {}
}
```

在组件模板中可以像下面这样使用 `Unless` 指令：

```
<p *myUnless="boolValue">
  myUnless 的值是 false。
</p>
```

```
<p *myUnless="!boolValue">  
  myUnless 的值是 true。  
</p>
```

当 boolValue 值为 false 时，段落一中的内容会被添加到 DOM 树中，即在页面中展示“myUnless 的值是 false。”；当 boolValue 值为 true 时，页面中展示内容变为“myUnless 的值是 true。”。

8.3.2 模板标签与星号前缀

本章 8.3.1 节实现了一个自定义的 Unless 结构指令，当表达式的值为 false 时，指令会渲染模板内容；当值为 true 时，会将模板内容从 DOM 树中移除。在指令的使用过程中，有两个值得关注的内容在上一节并未涉及——模板标签（<ng-template>）和星号前缀（*）。本节会就这两个关键内容进一步讲解。

模板标签

结构指令在将模板内容添加至 DOM 树和从 DOM 树中移除时，使用了 <ng-template> 标签。

在 <ng-template> 中定义的内容，默认的 CSS 样式 display 的属性值为 none，<ng-template> 标签中定义的脚本代码不会被执行，图片不会被加载，标签中的元素也不能被类似于 getElementById() 的方法访问。

在普通的 HTML 代码中，可以这样使用 <ng-template> 标签：

```
<p>Hello</p>  
<ng-template><p>Hi</p></ng-template>  
<p>World</p>
```

将上述代码片段分别在 Angular 环境和普通的 HTML 页面中使用，两者在浏览器中的显示是一样的，通过使用 Chrome 开发者工具查看所生成的 DOM 树，可以看到在 HTML 页面中，<ng-template> 标签内容被 #document-fragment 所包装；而在 Angular 环境中，<ng-template> 标签和内容被移除，在 <ng-template> 标签处仅留下了一行注释，如表 8-3 所示。

表 8-3 标签在不同环境中生成的 DOM 节点比较

浏览器显示	HTML 在浏览器中的源码	Angular 在浏览器中的源码
Hello World	<p>Hello</p><ng-template>#document-fragment<p>Hi</p></ng-template><p>World</p>	<p _ngcontent-jrg-1>Hello</p><!--template bindings={} --><p _ngcontent-jrg-1>World</p>

星号前缀

在上一节实现的 Unless 指令中，并没有使用 <ng-template> 标签，细心的读者应该已经发现，在 Unless 指令前面有一个星号 “*” 作为前缀。示例代码如下：

```
<p *myUnless="boolValue">
  myUnless 的值是 false。
</p>
```

星号是使用结构指令的语法糖，使用星号前缀可以简化对结构指令的使用，Angular 会将带有星号的指令引用替换成带有 <ng-template> 标签的代码。对比一下用星号前缀和 <ng-template> 标签方式来使用 Unless 指令。示例代码如下：

```
<!-- 使用 “*” 星号方式 -->
<p *myUnless="condition" > “*” 星号方式中的 myUnless </p>

<!-- 使用 `<ng-template>` 标签 -->
<ng-template [myUnless]="condition">
  <p>template 标签中的 myUnless</p >
</ng-template>
```

上述两种使用方式效果是一样的，Angular 会将使用星号前缀的方式转换成 <ng-template> 标签方式。它将元素及其内容放在 <ng-template> 标签内部，将指令移到 <ng-template> 标签中，用方括号括起来作为一个属性绑定。在组件中绑定给指令表达式的结果是一个布尔类型值，Angular 根据这一布尔值决定模板内容是否被渲染。

为了加深理解，再看看另一个常用结构指令 NgFor 的使用方式。示例代码如下：

```
<!-- 使用 “*” 星号方式 -->
<div *ngFor="let contact of contacts">{{ contact }}</div>

<!-- 使用 `<ng-template>` 标签 -->
<ng-template ngFor let-contact [ngForOf]="contacts">
```

```
<div>{{ contact }}</div>
</ng-template>
```

从上面示例可以看出，Angular 对星号前缀使用的转换方式基本类似。首先创建一个 `<ng-template>` 标签，然后将模板内容放到 `<ng-template>` 标签内部，并将指令移至 `<ng-template>` 标签中。在标签中，`ngFor` 是指令的选择器，`[ngForOf]` 绑定指令的输入属性，示例中将 `contacts` 集合绑定给 `[ngForOf]` 作为指令的输入，`let-contact` 创建了一个模板局部变量 `contact`。

8.3.3 NgIf 指令原理

上文中实现的 `Unless` 是一个和 `NgIf` 作用相反的指令，在应用开发中，`NgIf` 是一个使用频率很高的指令，它根据输入变量的布尔值，动态地在 DOM 树中插入或者删除目标元素的模板内容。

作为 Web 开发者，通常会选择将元素样式属性 `display` 设为 `none` 来隐藏目标元素。采用这种方式，这些元素虽然不可见，但却仍然保存在 DOM 中。这样带来的好处是，如果元素不久需要再次显示，组件不需要重新被初始化，组件的状态因为之前被保留，所以可以马上显示。但是将元素隐藏也会带来别的问题。在 Angular 应用里，如果隐藏一个元素，它仍然保留在 DOM 树中，Angular 会继续检测那些可能发生变化的数据绑定，组件的所有行为将会保持，在这种情况下，组件及其所有的子节点仍然会占用资源，消耗更多的内存，从而影响性能。

使用 `NgStyle` 指令可以通过改变样式将元素隐藏。当在组件中使用 `NgStyle` 指令进行如下设置时，可以将组件在 DOM 树中隐藏。示例代码如下：

```
<div [ngStyle]="{'display':'none'}">display none</div>
```

`NgIf` 指令不是通过改变样式将元素隐藏的，而是根据输入变量的布尔值，当输入变量值为 `false` 时，把这些元素从 DOM 树中移除，停止监测相关组件绑定的属性是否有变化，释放它的 DOM 事件监听器并且销毁组件，组件将会被垃圾回收并且释放内存。

上文中实现了 `Unless` 指令的代码逻辑，它的实现很简单，这里来看一下 `NgIf` 指令的实现。示例代码如下：

```
import { Directive, TemplateRef, ViewContainerRef } from '@angular/core';
import { isBlank } from '../facade/lang';

@Directive({selector: '[ngIf]', inputs: ['ngIf']})
export class NgIf {
  private _prevCondition: boolean = null;
```

```

constructor(
  private _viewController: ViewControllerRef,
  private _templateRef: TemplateRef<Object>
) {}

set ngIf(newCondition: any /* boolean */) {
  if (newCondition && (isBlank(this._prevCondition) || !this._prevCondition)) {
    this._prevCondition = true;
    this._viewController.createEmbeddedView(this._templateRef);
  } else if (!newCondition && (isBlank(this._prevCondition) || this._prevCondition)) {
    this._prevCondition = false;
    this._viewController.clear();
  }
}
}
}

```

在上述代码中，在 `@Directive` 装饰器中有两个键值对，首先为 `selector` 属性赋值，设置指令的选择器为 `[ngIf]`，然后将 `inputs` 属性赋值为 `['ngIf']`，为指令绑定了一个输入属性，它的值和指令选择器的值相同。

在 `Unless` 指令代码中，指令通过 `@Input()` 装饰器绑定了输入属性；在上述代码的 `@Directive` 装饰器中，以键值对形式为 `inputs` 元数据赋值，这是指令绑定输入属性的另一种方式，两者的效果是一致的。

`NgIf` 实现类的代码与 `Unless` 指令代码很相似，不同点在于，在 `NgIf` 中设置的 `_prevCondition` 变量用于记录上次表达式的值，并在自定义输入属性的 `set()` 方法中，对 `_prevCondition` 变量的值做变更检查，这样输入属性每次变更时都可以避免不必要的操作，从而提高性能。

回到本章“指令分类”节中使用 `NgIf` 结构指令的例子：

```

<!-- 示例 A -->
<p *ngIf="condition">
  condition 的值为 true，读者能看到这句话。
</p>

<!-- 示例 B -->
<p *ngIf="!condition">

```


!condition 的值为 false，读者不能看到这句话。
</p>

在 Angular 应用中，当 condition 值为 true 时，示例 A 中 ngIf 绑定的表达式结果为 true，示例 B 中表达式结果为 false。使用 Chrome 开发者工具查看所生成的 HTML 代码，如下所示：

```
<!--bindings={
  "ng-reflect-ng-if": "true"
}-->
<p _ngcontent-c0>
  condition 的值为 true，读者能看到这句话。
</p>
<!--bindings={
  "ng-reflect-ng-if": "false"
}-->
```

在所生成的 HTML 代码中，首先是一段由 Angular 自动插入的注释，然后是最终生成的 DOM 元素。示例 A 中模板元素内容能在 DOM 中渲染出来，是因为其绑定的表达式值为 true；而示例 B 中模板元素内容在 DOM 中被相应地删除，是因为其绑定的表达式值为 false。

8.4 扩展阅读

在本章 8.1.2 节中提到过一些使用场景相对较少的指令，本节将对这些生僻指令进行讲解。

NgTemplateOutlet 指令

NgTemplateOutlet 指令可以在模板中创建内嵌视图。

使用 NgTemplateOutlet 指令，需要为指令绑定一个对模板元素的引用。NgTemplateOutlet 指令的基本语法如下：

```
<ng-container *ngTemplateOutlet="tmplRef"></ng-container>
```

被引用的模板内容为：

```
<ng-template #tmplRef><span>Hello</span></ng-template>
```

除此之外，还可以为被插入的内嵌视图绑定一个上下文对象。可以将当前视图获取的上下文对象绑定到被插入的内嵌模板中，使用绑定上下文对象的语法如下：

```
<ng-container *ngTemplateOutlet="tplRef; context: myContext"></ng-container>
```

假设 myContext 的值为 { name: 'World' }, 那么被引用的 template 可以用 let 语法使用这个值。示例代码如下:

```
<ng-template #tplRef let-thename="name"><span>Hello {{thename}}</span></ng-template>
```

如果 let-thename 没有指定值, 那么默认读取 context 对象的 \$implicit 值。假设将 myContext 的值修改为 { \$implicit: 'World' }, 那么 template 可以改成如下所示:

```
<ng-template #tplRef let-thename><span>Hello {{thename}}</span></ng-template>
```

最终都是输出 Hello World。

NgPlural、NgPluralCase 指令

NgPlural 和 NgPluralCase 是一组搭配使用的指令。NgPlural 是一个国际化指令, 其作用和 NgSwitch 指令类似, 可以看作 NgSwitch 指令的一个变种; 而 NgPluralCase 指令可以类比 NgSwitchCase 来理解, 使用 NgPluralCase 指令的元素会根据匹配结果来展示。使用 NgPlural 指令, 会在元素子节点中寻找与 [ngPlural] 属性绑定值匹配的元素, 当匹配失败时, 将继续寻找与绑定值匹配的其他元素。

NgPlural 与 NgSwitch 指令的差异体现在: 使用 NgPlural 指令需要继承 NgLocalization 类并实现 getPluralCategory() 方法, 在这个方法中, 根据具体的分类需求, 将落在某一范围的值命名为一个分类, 并将分类名称返回。而返回的分类, 可以和 [ngPlural] 绑定值进行匹配, 当值属于这一分类的定义范围时, 就当匹配成功。

根据上述逻辑可以看出, NgSwitch 指令只能进行精确匹配; 而 NgPlural 指令除可以进行精确匹配外, 还可以进行范围匹配。

NgPlural 指令的匹配细则如下:

- 如果元素的 [ngPluralCase] 绑定值以 = 为前缀, 需要 [ngPlural] 绑定值与 [ngPluralCase] 绑定值精确匹配时才展示该元素。
- 否则, 该元素被认为进行“分类匹配”。此时, 在与 [ngPlural] 绑定值精确匹配的元素没有找到, 而绑定值又匹配了该分类的规则时, 则展示该元素。
- 如果精确匹配和分类匹配都没找到, 那么标记为 [ngPluralCase]="other" 的元素会作为默认项被匹配, 这一默认匹配规则相当于 NgSwitchDefault 指令的作用。

NgPlural 指令的使用示例如下：

```
class MyLocalization extends NgLocalization {
  getPluralCategory(value: any) {
    if(value < 5) {
      return 'few';
    }
  }
}

@Component({
  selector: 'app',
  template: `
    <p>Value = {{value}}</p>
    <button (click)="inc()">Increment</button>
    <div [ngPlural]="value">
      <ng-template ngPluralCase="=0">这是 0</ng-template>
      <ng-template ngPluralCase="=1">这是 1</ng-template>
      <ng-template ngPluralCase="few">这是 few</ng-template>
      <ng-template ngPluralCase="other">这是其他</ng-template>
    </div>
  `,
  directives: [NgPlural, NgPluralCase],
  providers: [provide(NgLocalization, {useClass: MyLocalization})]
})
export class App {
  value = 'init';

  inc() {
    this.value = this.value === 'init' ? 0 : this.value + 1;
  }
}
```

在上述示例中，首先定义了 MyLocalization 类，并实现了继承自 NgLocalization 类的 getPluralCategory() 方法，将小于 5 的值命名为 few 分类，随后在 @Component 装饰器中将 MyLocalization 类注入。在组件模板内容中，分别为几个元素的 ngPluralCase 属性绑定了不同的表达式，其中前两个为精确匹配，第三个为分类匹配，最后一个为默认匹配项。

FormGroupName 指令

FormGroupName 指令可以将一个已有的表单组合绑定到一个 DOM 元素上，它仅作为 FormGroupDirective 指令的子元素使用。

在下面示例中，将已有的表单组合 name 通过 FormGroupName 指令绑定到了表单的子元素 div 上，并且指定了绑定别名为 name。示例代码如下：

```
@Component({
  selector: 'my-app',
  template: `
    <div>
      <h2>Angular FormGroup 例子</h2>
      <form [formGroup]="myForm">
        <div formGroupName="name">
          <h3>输入你的名字</h3>
          <p>姓氏: <input formControlName="last"></p>
          <p>名字: <input formControlName="first"></p>
        </div>
        <h3>姓名: </h3>
        <pre>{{ myForm.get('name') | json }}</pre>
        <p>姓名是 {{myForm.get('name')?.valid ? "合法" : "不合法"}}</p>
        <h3>你喜欢的股票? </h3>
        <p><input formControlName="stock"></p>
        <h3>股票代码</h3>
        <pre> {{ myForm | json }} </pre>
      </form>
    </div>
  `,
})
export class App {
  myForm = new FormGroup({
    name: new FormGroup({
      first: new FormControl('', Validators.required),
      last: new FormControl('', Validators.required)
    }),
    stock: new FormControl()
  });
}
```

FormArrayName 指令

FormArrayName 指令可以将一个已有的表单数组绑定到一个 DOM 元素上，它仅作为 FormGroupDirective 指令的子元素使用。

在下面示例中，将已有的表单数组 cityArray 通过 FormArrayName 指令绑定到了表单的子元素 div 上，并且指定了绑定别名为 cities。

```
@Component({
  selector: 'my-app',
  template: `
    <div>
      <h2>Angular FormArray 例子</h2>
      <form [formGroup]="myForm">
        <div formArrayName="cities">
          <div *ngFor="let city of cityArray.controls; let i=index">
            <input [formControlName]="i">
          </div>
        </div>
      </form>
      <!--{cities: ['深圳', '广州']}-->
      {{myForm.value | json}}
    </div>
  `
})
export class App {
  cityArray = new FormArray([
    new FormControl('深圳'),
    new FormControl('广州')
  ]);
  myForm = new FormGroup({
    cities: this.cityArray
  });
}
```

8.5 小结

本章首先讲述了指令的类型。在 Angular 中，有属性指令、结构指令和组件三种指令。然后介绍了 Angular 常用的内置指令，根据使用场景的不同，可以分为通用指令、

路由指令和表单指令，并详细讲解了各个内置指令的使用方法，同时提到在 Angular 中，各类指令被包含在不同的 NgModule 模块中，应当以引用对应模块的方式来引入所需要的指令。接下来讲述了除内置指令外，还可以方便地自定义属性指令和结构指令。在自定义属性指令时，可以引用 DOM 元素，为指令绑定输入属性，同时可以响应用户的操作。在自定义结构指令时，可以根据输入的相应的值，动态地为组件添加和删除内容。为了进一步理解指令，本章还讲解了使用星号前缀和 `<ng-template>` 标签的原理。最后通过对 NgIf 指令源码的阅读，理解了指令实现的原理。在文中的“扩展阅读”部分还增加了对生僻指令的讲解，方便读者对指令有更为全面的认识。

通过本章的学习，相信读者对 Angular 的指令部分已经有了全面的了解。接下来，我们进入下一章的学习吧。

服务与 RxJS

在 Angular 中，服务用于帮助开发者书写可重用的公共功能（如日志处理、权限管理等）和复杂的业务逻辑，对于应用程序的模块化有着很重要的意义。

本章首先会介绍 Angular 服务的概念、优点，以及如何创建和使用服务，然后介绍一个非常常用的内置服务，即用于和远程服务器通信的 HTTP 服务。HTTP 服务是基于 RxJS 异步库（基于响应式编程范式实现）编写的，为了能更好地理解 HTTP 服务，本章还会详细讲解 RxJS 及其背后的响应式编程理念。

9.1 Angular 服务

9.1.1 概述

Angular 服务一般是封装了某种特定功能的独立模块，它可以通过注入的方式供外部调用。服务在 Angular 中的使用十分广泛，例如：

- 当多个组件中出现重复代码时，把重复代码提取到服务中实现代码复用。
- 当组件中掺杂了大量的业务代码和数据处理逻辑时，把这些逻辑封装成服务供组件使用，组件只负责与 UI 相关的逻辑，有利于后续的更新和维护。
- 把需要共享的数据存储在服务中，通过在多个组件中注入同一个服务实例实现数据共享。

以下是 Angular 应用中常见的几种服务：

- 和服务通信的数据服务。
- 检查用户输入的验证服务。
- 方便跟踪错误的日志服务。

9.1.2 使用场景

下面通过两个例子来学习 Angular 服务在业务逻辑封装和服务实例共享两种场景下的使用。

业务逻辑封装

下面结合通讯录例子中编辑联系人的功能来说明如何编写一个服务。

首先来分析编辑联系人都需要哪些步骤：

- 从服务器拉取联系人信息。
- 验证用户修改的数据。
- 把修改后的数据提交到服务器。

虽然开发者可以把所有的代码都写在组件里，但是这样会使组件的代码量非常大而且显得杂乱不堪，不利于后续代码的维护。所以，最好把从服务器拉取联系人信息和提交数据到服务器的代码封装到一个名为 `ContactService` 的类中。示例代码如下：

```
// contact.service.ts
import { Injectable } from '@angular/core';
// ...

@Injectable()
export class ContactService {

  // 从服务器上获取联系人信息
  getContactsData() {
    // ...
  }

  // 更新联系人信息到服务器
  updateContacts(contact) {
    // ...
  }
}
```




`@Injectable()` 装饰器用于说明被装饰的类依赖了其他服务，而这里 `ContactService` 没有依赖其他服务，所以 `@Injectable()` 是可以省略的，但是 Angular 官方推荐无论是否依赖了其他服务，都使用 `@Injectable()` 来装饰服务，因为添加 `@Injectable()` 装饰器有利于提高代码的可读性、一致性，减少异常的产生。

关于 `@Injectable()` 的更多细节可参阅第 10 章。

在通讯录例子的 `EditComponent` 组件中，通过依赖注入使用 `ContactService` 服务，需先将 `ContactService` 服务通过 `import` 导入，再在组件的构造函数中引入服务的实例，接着就可以在逻辑代码中调用服务的方法了。示例代码如下：

```
import { Component, OnInit, Input } from '@angular/core';
import { ContactService } from 'shared/contact.service';
@Component({
  selector: 'my-operate',
  templateUrl: 'app/edit/edit.component.html',
  styleUrls: ['app/edit/edit.component.css']
})
export class EditComponent implements OnInit {
  constructor(
    // ...
    private _contactService: ContactService,
    // ...
  ) {}
  // ...
}
```



这里没有在 `@Component` 的 `providers` 元数据上显式声明 `ContactService` 服务，是因为通讯录例子的服务采用的是模块注入的方式。在模块中注入服务，该模块下的所有组件都共享这个服务。可以在 `app.module.ts` 中看到 `ContactService` 的配置。

共享服务实例

在“组件”章节中已经介绍了组件间通信的几种情况，但在实际开发中，通常需要在多个组件之间进行通信，在这种情况下可以通过组件间共享同一个服务实例来实现通信。下面实现一个可在组件间共享数据的服务 `SharedService`。示例代码如下：

```
import { Injectable } from '@angular/core';

@Injectable()
export class SharedService {
  list: string[] = [];

  append(str: string){
    this.list.push(str);
  }
}
```

该服务包含一个 `list` 数组对象和一个 `append()` 方法，组件可以调用该服务的 `append()` 方法往 `list` 数组中添加数据。在下面例子中，父组件 `ParentComponent` 和子组件 `ChildComponent` 中都注入了 `SharedService` 服务。这两个组件要做的事情都很简单，子组件接收用户输入并调用 `SharedService` 的 `append()` 方法添加数据，父组件则把 `SharedService` 的数据变化实时展示到模板中。父组件的示例代码如下：

```
// 父组件

import { Component } from '@angular/core';
import { SharedService } from '../shared.service';
import { ChildComponent } from '../child.component';

@Component({
  selector: 'parent-component',
  template: `
    <ul *ngFor="let item of list">
      <li>{{item}}</li>
    </ul>
    <child-component></child-component>
  `,
  providers: [SharedService]
})
export class ParentComponent {
```

```
list: string[] = [];  
constructor(private _sharedService: SharedService) {}  
ngOnInit(): any {  
  this.list = this._sharedService.list;  
}  
}
```

需要注意的是，为了让父组件和子组件能获取到 `SharedService` 的同一个实例，需要在父组件中添加 `providers: [SharedService]`，子组件不需要添加；否则，父组件和子组件获得的 `SharedService` 将是两个不同的实例。也可以在父组件和子组件所属的模块中统一配置 `providers: [SharedService]`，那么父组件就不需要配置了。

子组件的示例代码如下：

// 子组件

```
import { Component } from '@angular/core';  
import { SharedService } from '../shared.service';  
  
@Component({  
  selector: 'child-component',  
  template: `  
    <input type="text" [(ngModel)]="inputText"/>  
    <button (click)="add()"></button>  
  `,  
})  
export class ChildComponent {  
  inputText: string = 'Testing data';  
  constructor(private _sharedService: SharedService) {}  
  add() {  
    this._sharedService.append(this.inputText);  
    this.inputText = '';  
  }  
}
```

在父子组件中注入同一个服务实例是实现本节示例的关键，也是层级注入的一个应用场景。关于层级注入的更多内容请参阅第 10 章。

9.2 HTTP 服务

前一节介绍了 Angular 服务的概念、优点，以及创建和使用服务的方法。接下来将会学习 Angular 中最常用的服务——HTTP 服务。HTTP 服务是 Angular 中使用 HTTP 协议与远程服务器进行通信而提供的一系列服务，被封装成独立模块。截至本书写作之时，Angular 提供了两个模块封装 HTTP 服务，分别是 `HttpModule`（存放在 `@angular/http` 包中）和 `HttpClientModule`（存放在 `@angular/common/http` 包中）。`HttpClientModule` 是 Angular 4.3 版本之后推出的，是 `HttpModule` 的增强版本，而 `HttpModule` 很有可能会在后面的版本中被移除。所以，如果项目是基于 Angular 4.3 及以上版本开发的，推荐使用 `HttpClientModule`。

9.2.1 HttpModule

`HttpModule` 模块主要包含了以下常用服务。

- `Http`：封装了常用的 HTTP 请求方法。
- `Headers`：封装了 HTTP 请求头。
- `RequestOptions`：封装了 HTTP 请求参数，它有一个子类 `BaseRequestOptions`，默认将请求方法设置为 GET。
- `ResponseOptions`：封装了 HTTP 响应参数，它有一个子类 `BaseResponseOptions`，默认将响应设置为成功。

在实际开发中接触比较多的是 `Http` 服务，它做了很多封装，使得开发者可以很方便地向后台发送 HTTP 请求。

使用 `HttpModule` 里的服务只需要三个简单的步骤。

- （1）在模块装饰器 `@NgModule` 中导入 `HttpModule`。
- （2）在组件模块中导入 `Http` 服务。
- （3）在组件的构造函数中注入服务实例。



上文提及的 `SharedService` 服务是通过 `@Component` 里的 `providers` 属性（元数据）注入组件中的（即在组件中注入服务），而上述 `Http` 服务则是通过导入 `HttpModule` 注入模块中的（即在模块中注入服务）。通过这两种方式注入，服务都能被组件正常使用，但这两种注入方式形成的作用域范围会有差异，详细内容可参见第 10 章。

下面是调用 Http 服务的一个例子。示例代码如下：

```
// app.module.ts
import { HttpClientModule } from '@angular/http';
// ...

@NgModule({
  imports: [
    HttpClientModule // 1. 在 @NgModule 中导入 HttpClientModule
  ],
  // ...
  bootstrap: [ AppComponent ]
})
export class AppModule {}

// contact.component.ts
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser/browser';
import { Http } from '@angular/http'; // 2. 导入Http 服务

@Component({
  selector: 'contact',
  template: '<div>hello http service!</div>'
})
export class ContactComponent {
  constructor(http: Http){ // 3. 引入 Http 服务
    // ...
  }
}
```

引入 Http 服务后，组件就可以用 AJAX 和 JSONP 两种方式发送数据请求了，下面将分别进行介绍。

AJAX 介绍

AJAX（Asynchronous JavaScript and XML）是使用 XMLHttpRequest 对象向服务器发送请求并处理响应的通信技术。XMLHttpRequest 支持以同步或异步的方式发送请求，同步的方式往往会造成页面“假死”，为了提升用户体验，通常使用异步的方式发送 HTTP 请求。

一般有三种方式处理异步操作，下面依次说明。

使用回调函数

使用回调函数是最基本的异步操作方式，形如 `functionB(functionA)`，一般是一个函数作为另一个函数的入参的形式，在执行 `functionB` 的时候把 `functionA` 作为参数传进去并在 `functionB` 里面调用 `functionA`，以此来实现回调功能。这种方式简单且易于理解，但容易造成冗长的回调链（多层嵌套，也叫回调地狱）问题，不利于代码维护，这里就不以此举例了。

使用 Promise

Promise 给异步操作提供了统一的接口，使得程序具备正常的同步运行流程，回调函数也不必再层层嵌套了，最终代码更易于理解、可维护性强。使用 Promise 的操作方式的示例代码如下：

```
// Promise 的写法
(new Promise(function (resolve, reject) {}))
  .then(funcA)
  .then(funcB)
  .then(funcC);
```

使用 Observable

使用 Observable 处理异步操作是 Angular 推荐的方式，HTTP 服务的 API 接口返回的也是 Observable 对象。

Observable 是响应式编程模型 Rx 的核心概念。Rx 的全称是 Reactive Extension，是微软开发的一套响应式编程模型，RxJS 则是它的 JavaScript 版本。Angular 对 RxJS 做了封装处理，使得在 Angular 开发中更方便使用。本节的示例代码会简单展示 RxJS 的使用，要深入了解 RxJS，请阅读下一节“响应式编程”部分。

下面将介绍用 HTTP 服务发起 GET 及 POST 请求，让读者更直观地了解如何使用 HTTP 服务和 RxJS 实现与服务器的交互。

GET 请求

获取服务器数据是 HTTP 服务使用最多的场景，现在结合通讯录例子来看看如何编写一个提供联系人信息的数据服务。示例代码如下：

```
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/http';
import { AppComponent } from './app.component';
```

```
@NgModule({
  imports: [
    HttpClientModule
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

首先在 `@NgModule` 装饰器中引入 `HttpClientModule`，后面编写组件时就不需要在 `providers` 数组中引入了。数据服务一般包含增、删、查、改等功能，这里从最简单的查询联系人开始讲解，示例代码如下：

```
// contact-old.service.ts
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import "rxjs/add/observable/throw";
import "rxjs/add/operator/map";
import "rxjs/add/operator/catch";
```

```
const CONTACT_URL = `/assets/contacts.json`;
```

```
@Injectable()
export class ContactService {
  constructor(private http: Http) {}

  getContactsData() {
    return this.http.get(CONTACT_URL)
      .map(this.extractData)
      .catch(this.handleError);
  }

  private extractData(res: Response) {
    let body = res.json();
    return body || {};
  }

  private handleError (error: any) {
```

```

    let errMsg = (error.message) ? error.message :
      error.status ? `${error.status} - ${error.statusText}` : 'Server error';
    console.error(errMsg); // 打印到控制台
    return Observable.throw(errMsg);
  }
}

```

正如前文所说的，`this.http.get()` 返回的是一个 `Observable` 对象，而 `map()` 方法对数据进行转换。在 `extractData()` 方法里需要通过 `json()` 方法把服务器返回的数据转换成 JSON 对象。

也许读者会认为在 `getContactsData()` 方法中直接返回 `Response` 对象更加方便，但是不建议这样做。数据服务应该对使用者隐藏实现的细节，使用者只需要调用数据服务的接口取得数据即可，并不需要关心数据是如何获得的，以及是何种数据格式。

最后不要忘记处理异常情况，任何 I/O 操作都有发生异常的可能（如网络故障等），所以在数据服务里做好异常处理是十分必要的。在这个例子里，通过 `catch` 操作符捕捉到错误并打印到控制台，然后使用 `Observable.throw()` 方法重新返回一个包含错误信息的 `Observable` 对象。

现在看看在组件中怎样使用 `ContactService` 这个数据服务，示例代码如下：

```

// list.component.ts:

import { Component } from '@angular/core';
import { ContactService } from 'shared/contact.service';

@Component({
  // ...
})

export class ListComponent {
  // ...
  constructor(
    private _contactService: ContactService
  ) {}
  getContacts() {
    return this._contactService.getContactsData()
      .subscribe(
        contacts => this.contacts = contacts,

```



```
        error => this.errorMessage = <any>error
    );
}
}
```

在 `ListComponent` 组件中调用了 `this._contactService.getContactsData()` 方法，同时使用 `subscribe()` 方法的第二个参数来处理错误信息。这里使用 `this.errorMessage` 变量来保存，可以通过将该变量绑定到模板上显示给用户。



需要注意的是，在 `ContactService` 的 `getContactsData()` 方法中，`http.get()` 并没有发出请求，因为 RxJS 中的 `Observable` 实现的是“冷”模式，只有当它被 `getContactsData().subscribe()` 订阅之后才会发送请求。关于冷热模式的概念会在后面详解。

POST 请求

上面的例子已经实现了从远程服务器拉取联系人数据，通讯录应用还应该添加和编辑联系人的功能。接下来在 `EditComponent` 组件里添加一个 `addContact()` 方法，其参数是联系人 `Contact` 的实例。示例代码如下：

```
// edit.component.ts
// ...

addContact (contact) {
    // do something
}
```

服务器端的接口是符合 REST 规范的，添加联系人和拉取联系人的 URL 路径是一样的。添加联系人使用 POST 方法，并且在请求体中新增 `Contact` 对象的联系人数据。下面是请求体数据格式的一个例子。

```
{
  "name": "Angular",
  "telNum": "13500000000",
  "address": "广东省深圳市",
  "email": "book@angular",
  "birthday": "1990/10/10",
  "collection": 0
}
```

服务器端的接口在接收到数据并验证通过后会生成一个唯一的 id 保存到数据库中, 然后以 JSON 格式返回带 id 的新联系人数据。了解了新增联系人接口的实现机制后, 可以快速实现添加联系人功能了。因为要发起 POST 请求, 并且在请求体中传递 JSON 格式的数据, 所以要设置 HTTP 请求头 Content-Type 的值为 'application/json'。

首先需要导入 Headers 和 RequestOptions 对象, 示例代码如下:

```
import { Headers, RequestOptions } from '@angular/http';
```

接下来在 ContactService 服务中新增一个 addContact() 方法, 示例代码如下:

```
// contact.service.ts
// ...
addContact (obj: Object = {}) {
  let body = JSON.stringify(obj);
  let headers = new Headers({ "Content-Type": "application/json" });
  let options = new RequestOptions({ headers: headers });

  return this.http.post(CONTACT_URL, body, options)
    .map(this.extractData)
    .catch(this.handleError);
}
// ...
```

Headers 是 RequestOptions 的一个属性, RequestOptions 作为第三个参数传递给 HTTP 服务的 post() 方法, 这样就可达到自定义请求头的目的。

即使 Content-Type 已经被指定为 JSON 类型, 服务器端也仍然只接收字符串, 所以在发送请求前, 先要用 JSON.stringify() 方法把联系人数据处理一下。

在组件中使用 addContact() 方法和使用 getContact() 方法一样, 示例代码如下:

```
// edit.component.ts
import { ContactService } from 'shared/contact.service';

// ...

addContact (contact) {
  if (!contact) { return; }
  this._contactService.addContact(contact)
    .subscribe(
      contact => this.contacts.push(contact),
```

```

        error => this.errorMessage=<any>error
    );
}

```

同理，在组件的 `addContact()` 方法中订阅了 `ContactService` 中 `addContact()` 方法返回的 `Observable` 实例，请求返回时就会把新联系人数据追加到 `contacts` 数组中，用于展示给用户。

另外，HTTP 服务返回的 `Observable` 对象可以方便地转换成 `Promise` 对象。下面是 `ContactService` 服务的 `Promise` 版本，示例代码如下：

```

import { Injectable } from '@angular/core';
import { Http, RequestOptions } from '@angular/http';
import { Observable } from 'rxjs/Observable';

const CONTACT_URL = '/assets/contacts.json';

@Injectable()
export class ContactService {
    constructor(private http: Http) {
    }

    getContactsData(): Promise<any[]> {
        return this.http.get(CONTACT_URL)
            .toPromise()
            .then(this.extractData)
            .catch(this.handleError);
    }

    private extractData(res: Response) {
        let body = res.json();
        return body || { };
    }

    private handleError (error: any) {
        let errMsg = (error.message) ? error.message :
            error.status ? `${error.status} - ${error.statusText}` : 'Server error';
        console.error(errMsg); // 打印到控制台
        return Promise.reject(errMsg);
    }
}

```

JSONP 请求

在 Web 开发中，有时候需要向与当前页面不同源的服务器发起 AJAX 请求，这时会发现该请求被浏览器阻止了，这就是常说的浏览器同源策略的访问限制。所谓“源”就是 URI 协议（Scheme）、主机名（Host）和端口（Port）这几部分的组合，当全部相同时才算是同源的。

如果服务器和浏览器都支持 CORS（Cross-Origin Resource Sharing）协议，则 AJAX 不受同源策略的访问限制。CORS 是一个 W3C 标准，全称是“跨域资源共享”，它需要浏览器和服务器同时支持。



在编写本书时，主流的浏览器（如 Chrome 54、IE 11）都已支持 CORS，详情请从 [caniuse](http://caniuse.com/##search=cors) 网站进行了解：<http://caniuse.com/##search=cors>。

如果浏览器不支持或服务器端不方便实施 CORS，则可以选择 JSONP 方案，它适用于任何浏览器。众所周知，`<script>` 标签请求资源并不会受同源策略的限制，其实 JSONP 就是利用 `<script>` 标签的特性来绕过同源策略的。使用 JSONP 的关键是利用 `<script>` 标签发起 GET 请求，在这个 GET 请求中传递 callback 参数给服务器端，然后服务器端返回一段 JavaScript 代码，一般以 callback 函数包裹着 JSON 数据的形式返回。当 `<script>` 标签的请求完成后就会自动执行这段代码，所以可以在预先定义好的全局方法 callback 中接收和处理 JSON 数据。值得注意的是，JSONP 只能发起 GET 请求，在需要发起 POST 请求的场景中并不适用。

HTTP 服务中包含了 JSONP 服务，下面是使用 JSONP 服务的一个例子。示例代码如下：

```
import { Injectable } from '@angular/core';
import { Jsonp, URLSearchParams } from '@angular/http';

@Injectable()
export class ContactService {
  constructor(private _jsonp: Jsonp) {}

  getContactsData () {
    let URL = 'http://www.others.com/contactcs';
    let params = new URLSearchParams();
    params.set('format', 'json');
```

```

params.set('callback', 'JSONP_CALLBACK');

return this._jsonp
  .get(URL, { search: params})
  .map(res => res.json())
  .subscribe(
    contacts => this.contacts = contacts,
    error => this.errorMessage = <any>error
  );
}
}

```

上面的代码很简单，首先在构造函数中注入 JSONP 服务，然后使用 URLSearchParams 对象构造请求参数，最后调用 JSONP 服务的 get() 方法发起请求。

拦截器实现（HttpInterceptor）

在一个 Angular 应用中，可能会有非常多的地方使用了 HTTP 服务来处理网络请求，开发者可能需要对所有请求做统一的处理，例如添加一些必要的 HTTP 自定义请求头、在后端返回某个错误时进行统一处理（如 401 错误码），以及统一在请求发出前显示“加载中”的状态并在请求返回后关闭该状态等。在这些情况下，就可以通过实现 ConnectionBackend 类并重写 createConnection() 方法来达到统一处理的目的。



在本章之后的内容中讲述的改造 HTTP 服务，也能满足这个场景，在实际开发中可按需选择合适的方式。

首先编写一个 HttpInterceptor 服务，在请求发送前后进行相应的处理。示例代码如下：

```

// http-interceptor.ts
import { Injectable } from '@angular/core';
import { Request, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class HttpInterceptor {

  beforeRequest(request: Request): Request {

```

```

    // 请求发出前的处理逻辑
    console.log(request);
    return request;
  }

  afterResponse(res: Observable<Response>): Observable<any> {
    // 请求响应后的处理逻辑
    return res.map((data)=> {
      console.log(data);
      console.log('我在请求后');
      return data;
    });
  }
}

```

接着实现 `ConnectionBackend` 抽象类，目的是封装 `XHRBackend` 服务，在 `XHRBackend` 创建 `XHRConnection` 实例前后进行相应的逻辑处理。示例代码如下：

```

// http-interceptor-backend.ts
import { Injectable } from '@angular/core';
import { ConnectionBackend, XHRConnection, XHRBackend, Request } from '@angular/
  http'
import { HttpInterceptor } from './http-interceptor';

@Injectable()
export class HttpInterceptorBackend implements ConnectionBackend {
  constructor(private httpInterceptor: HttpInterceptor, private _xhrBackend:
    XHRBackend) {}

  createConnection(request: Request): XHRConnection {
    let interceptor = this.httpInterceptor;

    // 在请求发出前，拦截请求并调用 HttpInterceptor 对象的 beforeRequest() 方法
    // 进行处理
    let req = interceptor.beforeRequest ? interceptor.beforeRequest(request) :
      request;

    // 通过 XHRBackend 对象创建 XHRConnection 实例
    let result = this._xhrBackend.createConnection(req);
  }
}

```

```

    // 在得到响应后，拦截并调用 HttpInterceptor 对象的 afterResponse 方法进行处
    理
    result.response = interceptor.afterResponse ? interceptor.afterResponse(result.
        response) : result.response;

    return result;
}
}

```

从 `HttpModule` 源码中可以看出，HTTP 服务默认是使用 `XHRBackend` 对象作为构造函数的第一个参数创建的。

```

// ...
export function httpFactory(xhrBackend: XHRBackend, requestOptions: RequestOptions)
    : Http {
    return new Http(xhrBackend, requestOptions);
}

// ...

```

为了使刚刚定义的 `HttpInterceptorBackend` 拦截生效，需要将创建 HTTP 服务时的第一个参数改为 `HttpInterceptorBackend` 对象，因此我们定义了一个新的 `httpFactory` 工厂方法。示例代码如下：

```

// http-factory.ts
import { RequestOptions, Http } from '@angular/http';
import { HttpInterceptorBackend } from './http-interceptor-backend';

export function httpFactory(httpInterceptorBackend: HttpInterceptorBackend,
    requestOptions: RequestOptions): Http {
    return new Http(httpInterceptorBackend, requestOptions);
}

```

最后在根模块中导入以上定义的服务即可。为了简单起见，假设以上创建的代码文件都放在与通讯录例子根模块（`app.module.ts`）文件相同的路径下，那么在根模块中导入以上服务的示例代码如下：

```

// app.module.ts

// ...
import { Http, RequestOptions } from '@angular/http'
import { HttpInterceptorBackend } from './interceptor/http-interceptor-backend'

```

```
import { HttpInterceptor } from './interceptor/http-interceptor';
import { httpFactory } from './interceptor/http-factory';

// ...
providers: [
  // ...
  HttpInterceptorBackend, HttpInterceptor,
  {provide: Http, useFactory: httpFactory, deps: [HttpInterceptorBackend,
    RequestOptions]}
]
// ...
```

完成以上步骤后，当通过 Angular 的 HTTP 服务发出任何一个 HTTP 请求时，在控制台中都能打印出 Request 对象和 Response 对象。

9.2.2 HttpClientModule

上一节介绍了关于 HttpClientModule 的常用功能。HttpClientModule 能实现的功能，HttpClientModule 也都可以实现。而且相比于 HttpClientModule，HttpClientModule 的实现更优雅，也提供了一些更强大的特性，例如：

- 默认 JSON 解析，无须再手动进行 JSON 转换。
- 更优雅的拦截器（Interceptor）支持。
- 支持进度事件（Progress Event）。

HttpClientModule 模块常用到的服务如下。

- HttpClient：封装常用的 HTTP 请求方法。
- HttpHeaders：封装 HTTP 请求头，该对象为不可变对象。
- HttpParams：封装请求参数，该对象为不可变对象。
- HttpRequest：封装请求发送阶段的上下文信息，该对象为不可变对象。
- HttpResponse：封装请求响应阶段的上下文信息，该对象为不可变对象。
- HttpEvent：HTTP 事件联合类型，可以指代 HttpResponse、HttpProgressEvent 等类型。



不可变对象表示每次修改该对象时都会返回新的实例，原来的实例不会发生变化。

使用 HttpClientModule 的步骤也很简单，只需要三步，以最常用的 HttpClient 服务的使用为例，示例代码如下：

```
import { HttpClientModule } from '@angular/common/http';
// ...

@NgModule({
  imports: [
    HttpClientModule // 1. 在 @NgModule 中导入 HttpClientModule
  ],
  // ...
  bootstrap: [ AppComponent ]
})
export class AppModule {}

// 模块下的任一组件
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http'; // 2. 导入 HttpClient 服务

@Component({
  selector: 'some-cmp',
  template: '<div>hello http service!</div>'
})
export class SomeComponent {
  constructor(http: HttpClient){ // 3. HttpClient 服务引入
    // ...
  }
}
```

这样，组件便获得了 HttpClient 服务的实例引用。HttpClient 服务提供了常见的 AJAX 请求 API，如 GET、POST 等，与 HttpModule 里的 Http 服务的使用方法基本一致。

GET 请求

把 HTTP 服务的 GET 请求例子改造成 HttpClient 的例子，示例代码如下：

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpResponse } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
```

```
const CONTACT_URL = '/assets/contacts.json';

@Injectable()
export class ContactService {
  constructor(private http: HttpClient) {}

  getContactsData(): Observable<any[]> {
    return this.http.get(CONTACT_URL)
      // .map(this.extractData)
      .catch(this.handleError);
  }

  // private extractData(res: Response) {
  //   let body = res.json();
  //   return body || {};
  // }

  // 错误处理更加清晰
  private handleError (err: HttpResponseError) {
    let errMsg;
    if (err.error instanceof Error) {
      // 客户端本身出错，这时请求通常还没发出
      errMsg = err.error.message;
    } else {
      // 请求已经发出，但返回非 200 的 HTTP 状态码
      errMsg = `${err.status} - ${err.statusText}, 详细错误: ${err.error}`;
    }
    console.error(errMsg); // 打印到控制台
    return Observable.throw(errMsg);
  }
}
```

从上述示例代码可以看出：

- HttpClient 的接口返回的也是 RxJS 的 Observable 类型的对象。
- 错误处理变得更加清晰，区分客户端错误和请求错误。
- HttpClient 服务不再需要手动进行 JSON 解析，它默认会进行 JSON 解包。

假如不希望以默认的 JSON 方式解包，HttpClient 也支持指定响应包格式，示例代码如下：

```
this.http.get(CONTACT_URL, { responseType: 'text' })
  .subscribe((res: string) => {
    console.log(res);
  });
```

`get()` 方法的第二个参数用来指定该请求的一些配置，如这里的 `responseType`，它可以指定响应包的解析方式。除了 `text` 值，它还支持更多的响应类型，如 `arraybuffer`、`blob` 等。

除此之外，`HttpClient` 也支持与 `Http` 相同的返回格式，示例代码如下：

```
this.http.get(CONTACT_URL, { observe: 'response' })
  .subscribe((res: HttpResponse) => {
    console.log(res.body);
  });
```

POST 请求

POST 请求跟 GET 请求的用法类似，示例代码如下：

```
this.http.post(CONTACT_URL, 'text body', { observe: 'response' })
  .subscribe((res: HttpResponse) => {
    console.log(res.body);
  });
```

可以看到，第二个参数的值即为 POST 请求发送的内容，如 `'text body'` 字符串。当前 POST 数据越来越多地使用 JSON 数据，`HttpClient` 支持 JSON 对象入参，而无须手动调用 `JSON.stringify()` 来序列化。示例代码如下：

```
let body = { data: 'json body' }; // JSON 对象
this.http.post(CONTACT_URL, body, { observe: 'response' })
  .subscribe((res: HttpResponse) => {
    console.log(res.body);
  });
```

除了 JSON 对象，它还支持更多的对象类型，如 `ArrayBuffer`。示例代码如下：

```
let body = new ArrayBuffer(10); // ArrayBuffer 类型
this.http.post(CONTACT_URL, body, { observe: 'response' })
  .subscribe((res: HttpResponse) => {
    console.log(res.body);
  });
```

JSONP 请求

使用 HttpClient 发送 JSONP 请求也非常方便，示例代码如下：

```
this.http.jsonp(CONTACT_URL, 'jsonpCallback')
  .subscribe(() => {
    console.log('done');
  })
```

可以看到，第二个参数为 jsonp 的回调函数名称。

自定义 headers

HttpClientModule 里提供了 HttpHeaders 类来处理 HTTP header 信息，用法与 HttpClientModule 提供的 Headers 类类似。

首先新增引入 HttpHeaders 类，示例代码如下：

```
import { HttpClientModule, HttpHeaders } from '@angular/common/http';
```

然后即可使用该类构造 HTTP header 信息，示例代码如下：

```
let headers = new HttpHeaders({
  'Accept-Charset': 'utf-8',
  'My-Custom-Header': 'custom header value'
});
this.http.get(CONTACT_URL, { headers: headers })
  .subscribe(data => {
    console.log(data);
  })
```

拦截器实现

相比于 HttpClientModule，HttpClientModule 原生支持拦截器。要实现拦截功能，首先要编写拦截处理逻辑，示例代码如下：

```
import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/
  common/http';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor() {}
```

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
    let clonedReq = req.clone({  
        setHeaders: {  
            'X-AUTH-TOKEN': 'xxx' // 添加权限认证 token  
        }  
    });  
    return next.handle(clonedReq);  
};  
}
```



`HttpRequest` 的实例方法 `clone()` 可复制一份新的实例，并依据入参对象更新新实例的数据。之所以调用 `clone()` 方法，是因为 `HttpRequest` 实例是不可变的（Immutable），即每次修改该对象时都会返回新的实例。

`setHeaders` 属性可追加 header 配置，等价于 `req.clone({headers: req.headers.set('X-AUTH-TOKEN', 'xxx')})`；`req.headers` 为 `HttpHeaders` 类型，同样也是不可变的类型数据，调用 `req.headers.set()` 方法返回的是更新后的全新的 `headers` 实例。

从上述代码可以看到，`AuthInterceptor` 这个服务类实现了 `HttpInterceptor` 接口的 `intercept()` 方法，`intercept()` 的第一个参数为 `HttpRequest` 请求实例，开发者可对该实例做一些修改操作，如本例的添加请求头 `X-Auth-TOKEN`。如果需要新增或修改请求参数，则使用 `params` 配置项即可。示例代码如下：

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
    let clonedReq = req.clone({  
        setParams: {  
            'key': 'xxx'  
        }  
    });  
    return next.handle(clonedReq);  
}
```

使用 `next.handle()` 方法注册新的 `HttpRequest` 实例，返回一个 `Observable` 数据流，方便继续添加后面的拦截逻辑。`HttpClient` 服务的拦截逻辑是链式调用的，支持添加多个拦截器服务。

接下来开始使用这个拦截类。这个类本质是一个服务，需要先注入到模块里以供全

局使用，并且约定使用 HTTP_INTERCEPTORS 作为注入的 token。示例代码如下：

```
// ...
import { HTTP_INTERCEPTORS } from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true } // 注入
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

注入时，需要配置 multi 配置项为 true，使得该 token（HTTP_INTERCEPTORS）支持多个服务实例，多个拦截器将会按照编写顺序依次执行，例如：

```
providers: [
  { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }, // 先执行
  { provide: HTTP_INTERCEPTORS, useClass: SecondInterceptor, multi: true }, // 后执行
  // ...
],
```

HttpClient 除可以处理请求拦截外，也可以处理响应拦截。响应拦截也是在服务里完成的，并且复用同一个接口。示例代码如下：

```
@Injectable()
class ResponseInterceptor implements HttpInterceptor {
  constructor() {}
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req).map(event => {
      if (event instanceof HttpResponse) {
        if (event.status === 401) {
          // 登录态过期
        }
      }
    });
  }
}
```

```

    }
    return event;
  });
}
}

```

`next.handle()` 方法返回的是 `HttpEvent Observable` 类型对象，有了这个 `HttpEvent` 类型的数据流，开发者即可监听多种事件类型，如响应事件等。如上例所示，触发的事件为 `HttpResponse` 响应实例。拿到这个实例，开发者即可修改该实例上的任何数据了，如修改响应数据。示例代码如下：

```

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  return next.handle(req).map(event => {
    if (event instanceof HttpResponse) {
      event = event.clone({ body: 'new body data' });
    }
    return event;
  });
}

```

有了拦截器功能，开发者可以在请求的生命周期内对请求对象做更多通用的定制操作，如接口缓存、请求日志和模拟接口数据返回等，这些修改对原有的代码逻辑几乎不会造成影响，并且如插件一般即插即用。

进度事件（Progress Event）

`HttpClient` 的另一个特色功能是支持进度事件，进度事件通常可用来跟踪文件的上传和下载。上传和下载都需要花费一定的时间，利用进度事件接口可方便地在页面上添加友好的进度提示。

要开启进度事件，需要新建 `HttpRequest` 实例，并配置 `reportProgress` 选项属性。示例代码如下：

```

const req = new HttpRequest('POST', '/upload', file, {
  reportProgress: true // 开启进度事件
});

this.http.request(req).subscribe(event => {
  // 响应下载事件
  if (event.type === HttpEventType.DownloadProgress) {
    console.log(event); // 输出示例: { type: 3, loaded: 31, total: 31 }
  }
});

```

```
    let percent = Math.round(100 * event.loaded / event.total);
    console.log(`文件下载进度: ${percent}% 。`);
  }
  // 响应上传事件
  if (event.type === HttpEventType.UploadProgress) {
    console.log(event); // 输出示例: { type: 1, loaded: 2, total: 2 }
    let percent = Math.round(100 * event.loaded / event.total);
    console.log(`文件上传进度: ${percent}% 。`);
  }
  // 响应处理完成事件
  if (event.type === HttpEventType.Response) {
    console.log(event.body);
  }
});
```



进度事件的每次触发都会引起变化监测执行，所以 `reportProgress` 选项仅在界面需要进度提示时才开启。

9.3 响应式编程

上面主要介绍了 Angular 中常用的 HTTP 服务，并且通过通讯录例子讲解了如何使用 HTTP 服务从服务器拉取数据，以及向服务器发送数据。同时还介绍了 Angular 官方推荐的 RxJS 和 HTTP 服务搭配使用的方式。接下来将深入介绍响应式编程的概念及 RxJS 库。

9.3.1 概述

响应式编程（Reactive Programming），在维基百科中的解释是：“一种面向数据流（Data Flow）和变化传播（Propagation of Change）的编程范式。”

首先来看看面向变化传播的编程。面向变化传播就是看最初的数据是否会随着后续对应变量的变化而变化，比如在命令式编程（Imperative Programming）中，当 B 的数值改变之后，C 的数值并没有随着 B 的数值的改变而改变，如图 9-1 所示。

而在响应式编程中，随着 B 的数值的改变，C 的数值也会随之变动，如图 9-2 所示。

同样，在大家熟悉的 MVVM 中，存在一种 M（Model）到 V（View）的绑定关系，如图 9-3 所示。

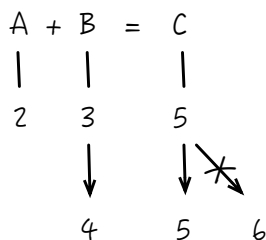


图 9-1 命令式编程数据变化

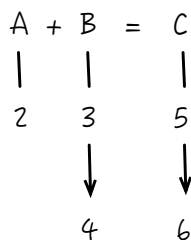


图 9-2 响应式编程数据变化

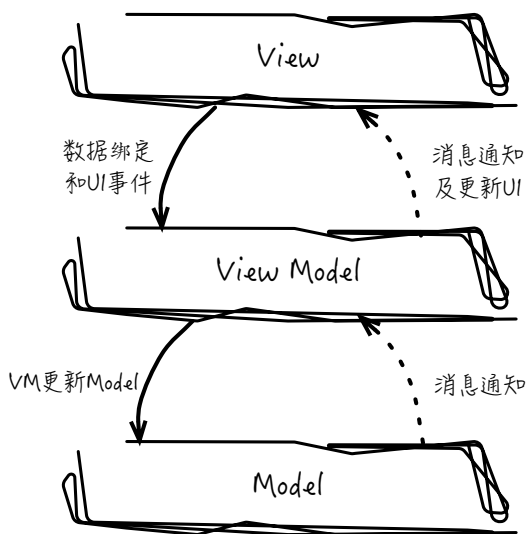


图 9-3 MVVM 模型示意图

当 Model 由 model1 变为 model2 时, View 也随之进行了变化, 由 view1 变为了 view2, 所以类似 MVVM 框架也体现了响应式编程中面向变化传播的思想。

介绍完面向变化传播, 再来看看面向数据流的编程。当监听一系列事件流并对这一系列事件流进行映射、过滤和合并等处理后, 再响应整个事件流的回调, 这个过程便属于面向数据流的编程。例如, 在下文将会介绍到的 ReactiveX 的编程范式中, 数据流被封装在一个叫作 Observable 的对象实例中, 通过观察者模式, 对数据流进行统一的订阅 (Subscribe), 并在中间插入像 filter() 这样的操作函数, 从而对 Observable 所封装的数据流进行过滤处理。示例代码如下:

```
myObservable.filter(fn).subscribe(callback);
```

通过这个例子, 可以看到响应式编程清楚地表达了动态的异步数据流, 而相关的计

算模型也自动地将变化的值通过数据流的方式进行了传播。

9.3.2 ReactiveX

ReactiveX (Reactive Extension), 一般简称为 Rx, 它是微软开发并维护的基于 Reactive Programming 范式实现的一套工具库集合, 于 2012 年 11 月开源, 用于提供一系列接口规范来帮助开发者更方便地处理异步数据流。Rx 系列结合了观察者模式、迭代器模式和函数式编程, 经过多年的验证, 目前已成为业界流行的响应式编程优秀实践。

Observable 介绍

在 Rx 中, 最核心的概念就是 Observable。应用中产生的异步数据都需要先包装成 Observable 对象, Observable 对象的作用是把异步数据转换为数据流的形式。所以所生成的这些 Observable 对象相当于数据流的源头, 后续操作都是围绕着这些被转换的流动数据展开的, 如图 9-4 所示。

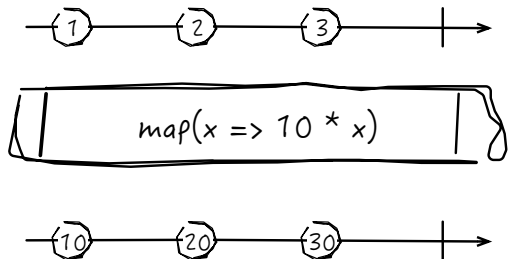


图 9-4 Rx 数据流动示意图

图 9-4 中最上面的箭头 (时间线) 代表了最初的 Observable 对象, 这个 Observable 数据流对外发出了 3 个数据, 这 3 个数据可能是 3 次点击事件携带的数据, 也可能是 3 次网络请求返回的数据。经过 map 操作后, 原来的 Observable 对象会变成一个新的 Observable 对象, 并且原来的 3 个数据会转换成新的数据在新的 Observable 对象数据流里流动。这样的操作实际上跟车间生产流水线非常相似, Observable 对象相当于半成品输入, 而上述 map 操作相当于流水线上的工人, 加工后最终输出成品, 而这个 map 操作在 Rx 里就称为 Operator。这样的操作更像是对一个事件集合做了过滤处理, 生成了一个新的事件集合。Rx 便是借鉴了集合的操作思想, 把复杂的异步数据流的处理问题, 简化成了同步的集合处理问题。换句话说, 在 Rx 中, 通过 Observable, 开发者可以像操作集合一样操作复杂的异步数据流。

Operator 介绍

Rx 在结合了观察者模式的同时，还结合了函数式编程和迭代器模式的思想。其中，Rx 的 Operator 便是对这两种编程模式的具体体现。

顾名思义，Operator 是 Rx 中 Observable 的操作符。在 Rx 中，每一个 Observable 对象，或者说数据流，都可以通过某个 operator 对该 Observable 对象进行变换、过滤、合并和监听等操作。同时，大多数的 operator 在对 Observable 对象处理后会返回一个新的 Observable 对象供下一个 operator 进行处理，这样方便在各个 operator 之间通过链式调用的方式编写代码。示例代码如下：

```
// 生成一个新的 Observable 对象
let newObservable = observable.debounceTime(500).take(2);
```

在上述代码中，debounceTime() 及 take() 都是一个 Operator，这些 Operator 通过链式方法的组合，对原有的 Observable 对象进行操作，最终返回了一个新的 Observable 对象。

在 Rx 中，Observable 作为观察者模式中的被观察者，需要一种方法来订阅它，而 subscribe() 便是这样的一种方法，订阅 Observable 对象发出的所有事件。示例代码如下：

```
observable.subscribe(observer);
```

subscribe() 方法会接收一个 observer 作为参数，来对 observable 发出的事件进行订阅。每当 observable 完成并发送（Emit）一个事件时，该事件就会被 observer 所捕获，进入 observer 对应的回调函数中。被 subscribe() 订阅过的 Observable 对象并不会返回一个新的 Observable 对象，因为 subscribe() 不是一个可以改变原始数据流的函数。相反，subscribe() 会返回一个 Subscription 实例，这个 Subscription 实例又提供了很多操作 API，例如具有取消订阅事件功能的 unsubscribe()，这里就不再做过多的展开介绍了。

其他核心概念

除了 Observable 及 Operator，Rx 中还有一些其他的核心概念，例如：

- Observer：对 Observable 对象发出的每个事件进行响应。
- Subscription：Observable 对象被订阅后返回的 Subscription 实例。
- Subject：EventEmitter 的等价数据结构，可以当作 Observable 被监听，也可以作为 Observer 发送新的事件。

由于篇幅有限，在这里就不对 Rx 中更多的概念进行展开讲解了，感兴趣的读者可以到 Rx 的官网进行更多的了解和学习。

9.4 RxJS

上面介绍了一些 Rx 的知识点，Rx 继承自响应式编程范式，已经在多种编程语言中实现。本节将要讲述的 RxJS 就是其在 JavaScript 层面上的实现，除此之外，还有 RxJava、Rx.Net、RxSwift 等。下面将从创建 Observable 对象开始介绍 RxJS。

9.4.1 创建 Observable 对象

本节将通过一个简单的例子来进行说明。首先把数据流封装为统一的 Observable 对象，并对它进行相应的处理。示例代码如下：

```
let button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click') // 返回一个 Observable 对象
  .subscribe(() => console.log('Clicked!'));
```

可以看到，通过 Observable 中的 fromEvent 静态方法（Static Method），把 <button> 标签的所有点击事件封装到一个 Observable 对象中，并转换为数据流的形式，最后通过 subscribe() 方法对整个点击事件流进行监听。也就是说，当按钮被点击时，对应的 Observable 对象便发出一条相应的消息，这条消息会被 subscribe() 中的 observer 回调函数所捕获，从而执行 console.log() 语句，这样便实现了一个最简单的数据流监听。

9.4.2 使用 RxJS 处理复杂场景

在介绍完第一个简单的例子之后，读者可能会有疑问：这不就是简单的事件监听吗？为什么还需要 Observable、Operator 这些复杂的概念呢？下面来看一个复杂点的例子。

在实际开发中，经常会遇到这样的场景：当用户在一个文本框中进行输入时，需要对用户的输入进行实时的监听，每当用户输入一些新的字符时，便会发一个请求到服务器端，来获取一些输入推荐信息展示给用户。但在实现这个功能的时候，需求可能还会有如下一些限制条件：

- 不必在每次用户输入的时候都发请求。用户在文本框中输入文字时，可能会输入得很快，这时是不需要给用户推荐任何信息的，不然频繁发送网络请求可能会给应用带来一些性能问题，因此可以做下优化。例如，当用户输入停顿 500ms 没有再输入时，才返回推荐信息给用户。

- 要保证请求返回的顺序。在异步请求的情况下，由于服务器返回推荐数据的响应时间会受网络环境等因素的影响，有时前端拿到的推荐数据不是最后一次请求的，所以需要保证这些推荐信息的渲染顺序要与请求顺序一致。

对于对用户输入进行监听并发起网络请求这样的需求，并不难实现，用原生的 JavaScript 通过监听和执行回调函数便可以解决。但是要满足上述两个额外限制条件，用原生的 JavaScript 实现会稍显复杂，在代码上需要新增各种各样的状态量，使得原本的逻辑变得冗余难读，同时这些状态量往往难以维护，久而久之便成了某些缺陷的源头。

然而在 RxJS 中，这样的数据流操作却可以实现得非常优雅，示例代码如下：

```
let inputSelector = document.querySelector('input');
Rx.Observable.fromEvent(inputSelector, 'keyup')
  .debounceTime(500)
  .switchMap(event => getRecommend(event.target.value))
  .subscribe(callback);
```

这段代码在后续章节中会有详细的讲解，这里不做过多的分析。从上述例子可以粗略看到，RxJS 借鉴了函数式编程的理念，把所有状态量的操作都封装在一个个的 Operator 函数中，并且通过链式调用的方法对原始数据流进行了过滤等操作，使得代码变得更易阅读和维护。

9.4.3 RxJS 和 Promise 的对比

前面提到，RxJS 的 Observable 可以通过 toPromise() 方法把原有的 Observable 对象转为 Promise 对象。那 RxJS 和 Promise 究竟谁优谁劣呢？这里先给出结论：能用 Promise 的场景 RxJS 都适用，RxJS 是作为 Promise 的超集存在的。

先来看看 Promise 实例的创建：

```
let promise = new Promise((resolve, reject) => {
  // ... some code

  if (/* 异步操作成功 */) {
    resolve(value);
  } else {
    reject(error);
  }
});
```

再来看看 Observable 实例的创建：

```
let Observable = new Observable(observer => {  
  observer.next(value1);  
  observer.next(value2);  
  
  observer.error(err);  
});
```

通过以上例子不难发现，Promise 只能针对单一的异步事件进行 resolve() 操作，而在 Observable 中，不仅能处理单一的异步事件（即调用 observer 的 next() 方法），而且能以流的形式响应多个异步事件。还有，对于 Promise 使用场景最多的 all()、race() 等方法，RxJS 同样有对应的解决方案。例如在 Promise 中，all() 方法被用来合并请求，示例代码如下：

```
let newPromise = Promise.all(promiseReq1, promiseReq2);
```

而在 Observable 中，有对应的 forkJoin() 方法来合并请求：

```
let newObservable = Rx.Observable.forkJoin(obsReq1, obsReq2);
```

同样，对于 race()，RxJS 也可以通过 merge() 和 take() 来实现。而对于其他一系列操作，RxJS 都有对应的解决方案，这里不做过多的赘述。综上所述，说 RxJS 是 Promise 的超集一点都不为过。

9.4.4 “冷”模式下的 Observable

在 Rx 的理念中，Observable 通常可以实现成“热”（Hot）模式或者“冷”（Cold）模式。在“热”模式下，Observable 对象一旦创建，便会开始发送数据。而在“冷”模式下，Observable 对象会一直等到自己被订阅，才会开始数据流的发送。在 RxJS 中，Observable 实现的是“冷”模式，示例代码如下：

```
console.log('Observable 的数据发送顺序为：');
```

```
let obs = new Observable(observer => {  
  console.log('Observable start');  
  observer.next();  
});
```

```
console.log('start');  
obs.subscribe();
```

代码执行的结果如下：

Observable 的数据发送顺序为：

start

Observable start

可以看到，在 RxJS 中，`observer.next()` 在 Observable 对象被订阅后才执行。也就是说，在 RxJS 中，Observable 对象直到被 `subscribe()` 方法订阅之后，才会进入数据发送的流程中。

除“冷”模式和“热”模式之外，RxJS 中还存在另外一种被称作 Connectable 的模式。在这种模式下 Observable 对象不管有没有被订阅，都不会发送数据，除非 ConnectableObservable 实例的 `connect()` 方法被调用。示例代码如下：

```
console.log('Connectable Observable 的数据发送顺序为：');
```

```
let obs = new Observable(observer => {  
  console.log('Observable start');  
  observer.complete();  
}).publish();
```

```
console.log('start');
```

```
obs.subscribe();
```

```
console.log('after Observable has been subscribed');
```

```
obs.connect();
```

代码运行后的结果如下：

Connectable Observable 的数据发送顺序为：

start

after Observable has been subscribed

Observable start

在上面的例子中，原来的 Observable 对象被 `publish()` 方法转为 Connectable 模式，在 Connectable 模式下，Observable 对象并没有在被 `subscribe()` 方法订阅之后发送数据，而是在被 `connect()` 方法调用后才发送的，这便是 Connectable Observable。

9.4.5 RxJS 中的 Operator

在对 RxJS 有了基本的了解后，再来讲讲最能体现 RxJS 强大之处的 Operator。

在 RxJS 中，Operator 操作符大抵可以分为：创建操作符、变换操作符、过滤操作符、组合操作符、工具操作符等，这里仅对常用的 Operator 稍作讲解。

创建操作符

由于前面一些讲解的需要，读者已经接触到少许 RxJS 中的创建操作符（Creation Operator），例如 `Observable.fromEvent()` 和 `new Observable()`。除了这些，经常用到的还有 `Observable.create()`，示例代码如下：

```
let observable = Rx.Observable.create(observer => {
  getData(data => {
    observer.next(data);
    observer.complete();
  })
});

observable.subscribe(data => {
  // doSomething(data);
});
```

在上面的例子中，`Observable.create()` 接受一个工厂函数（该函数以 `observer` 作为参数传入）并返回一个新的 `Observable` 对象。这个对象最终被 `subscribe()` 方法所监听，每当 `observer` 的 `next()` 方法被调用时，`subscribe()` 中的 `callback` 函数便捕获到 `observer` 传来的数据并进行相应的处理。这样便实现了对数据流的订阅和监听功能。

变换操作符

有些时候，通过 `Observable` 对象获取到的数据需要做一些批量的小调整。比如，数据获取的接口经常会用自己的一套规范来包裹数据，如下所示：

```
{
  "err_code": 0,
  "data": {"name": "Operators"}
}
```

只有数据中的 `data` 字段是实际想要处理的，所以需要对每一个请求做一次变换操作，把原本的数据流变换成所需要的数据流，这就需要用到 RxJS 的变换操作符（Transformation Operator）。RxJS 中最常用的变换操作符是 `Observable.prototype.map()`，用上个例子中的 `Observable` 对象来举例，示例代码如下：


```
observable.map(response => {  
  return response.data;  
}).subscribe(data => {  
  // doSomething(data);  
});
```

当 observable 拿到响应数据 response 并传给 observer 之前，可以通过 map 操作预先对 response 进行处理，从而让 observer 得到被加工后的数据格式。

过滤操作符

过滤操作符（Filtering Operator）可以用于过滤掉数据流中一些不需要处理的数据。回到上述例子，有时候在前端获取数据的时候，接口会因为各种原因无法返回最终需要的数据，可能由于异常导致返回的数据为空，或者只返回一个错误代码及错误描述告知前端，但是并不需要处理这些错误信息，所以需要过滤掉这些数据（或者根据实际情况对这些错误进行提示，不过这里不予考虑）。这时候就需要用到 Observable.prototype.filter() 对数据流进行过滤，示例代码如下：

```
observable.filter(response => {  
  return !!response.data && response.status === 200;  
}).map(response => {  
  return response.data;  
}).subscribe(data => {  
  // doSomething(data);  
});
```

通过 observable.filter() 对数据流进行过滤，结果为 false 的数据将不会再流向下一个 operator（observable.map）。

组合操作符

通过上面内容的学习，我们知道每个接口都可以转换为相应的数据流。很多业务场景需要依赖两个甚至更多的接口数据，并且在这些接口数据都成功获取后，再进行关联合并。要满足这样的业务场景，就需要把各个数据流汇合组成新的数据流，这时候就需要用到组合操作符（Combination Operator），常用的组合操作符有 Observable.forkJoin()。示例代码如下：

```
let getFirstDataObs = Rx.Observable.create(observer => {  
  observer.next(getFirstData());  
  observer.complete();  
});
```

```
});  
let getSecondDataObs = Rx.Observable.create(observer => {  
  getSecondData(data => {  
    observer.next(data);  
    observer.complete();  
  });  
});  
let observable = Rx.Observable.forkJoin(  
  getFirstDataObs, getSecondDataObs  
);  
  
observable.subscribe(datas => {  
  // datas[0] 是 getFirstDataObs 的数据  
  // datas[1] 是 getSecondDataObs 的数据  
  // doSomething(data);  
});
```

`Observable.forkJoin()` 把原本两个相互独立的 `Observable` 对象合并为一个新的 `Observable` 对象，它会在两个 `Observable` 对象的数据都抵达后才开始合并处理。所以本例中的 `doOtherthing()` 只会执行一次，此时拿到的 `datas` 便是包含两个数据流数据的数组。



`Observable.forkJoin()` 是静态函数，并不是实例函数（`Observable.prototype.forkJoin()`），使用时需要注意。部分操作符只有实例函数版本而没有静态函数版本，也有同时存在静态函数和实例函数两个版本的，它们的效果通常是一样的，至于开发者要使用哪个版本，看个人爱好。

如果某次数据请求需要依赖前一次请求的结果，也就是说，两次请求必须有先后顺序，这时候可以使用 `Observable.prototype.concatMap()`。示例代码如下：

```
let getFirstDataObs = Rx.Observable.create(observer => {  
  observer.next(getFirstData());  
  observer.complete();  
});  
  
let createSecondDataObs = function(firstData) {  
  return Rx.Observable.create(observer => {  
    getSecondData(firstData, secondData => {  
      observer.next(secondData);  
    });  
  });  
};
```

```
        observer.complete();
    });
});
}

let observable = getFirstDataObs.concatMap(firstData => {
    return createSecondDataObs(firstData);
}).subscribe(secondData => {
    doSomethingWithSecondData(secondData);
});
```

通过 `Observable.prototype.concatMap()` 方法, `getSecondDataObs()` 的数据流被紧接在 `getFirstDataObs()` 的数据流后, 并且最终数据流被 `subscribe()` 所捕获。

工具操作符

在 `Observable.prototype` 中有很多有用的工具方法, 统称为工具操作符 (Utility Operator), 像 `Observable.prototype.delay()` 或者 `Observable.prototype.timeout()` 等。当想要给某个请求设置 `timeout` 时, 就会用到这些方法。示例代码如下:

```
prevObservable.timeout(5000).subscribe(data => {
    doSomething(data);
}, err => {
    handleErr(err);
});
```

在这个例子中, 当 `prevObservable` 超过 5000ms 没有返回数据流时, 便会抛出一个 `err` 被 `subscribe()` 函数中的 `handleErr()` 方法所捕获。

9.4.6 Angular 中的 RxJS

在 Angular 中, RxJS 的应用随处可见, 例如: HTTP 服务中的 `get/post` 等方法会返回一个 `Observable` 对象, 以及路由中的 `events`、`params` 等也都是 `Observable` 对象。下面通过两个比较常见的场景, 来展现在 Angular 中如何灵活应用 RxJS 辅助开发。

数据处理

利用 RxJS, 开发者可以方便地实现错误处理和友好的加载提示。示例代码如下:

```
get(url: string) {
    this.showLoading(); // 开启加载动画
```

```

return this.http.get(url)
  .do(this.hideLoading.bind(this)) // 关闭加载动画
  .map(this.process.bind(this)) // 对返回数据进行处理
  .catch(this.handleError.bind(this)); // 处理异常情况
}

```

如果希望所有的请求都自动带上加载动画，或者做一些统一的异常处理，这时候可以结合 HttpClient 的拦截器来实现。示例代码如下：

```

@Injectable()
class CommonInterceptor implements HttpInterceptor {
  constructor() {}
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    this.showLoading(); // 开启加载动画
    return next.handle(req)
      .map(event => {
        if (event instanceof HttpResponse) {
          this.hideLoading(); // 关闭加载动画
          event = event.clone({
            body: this.preprocess(event.body) // 统一的数据预处理
          });
        }
        return event;
      })
      .catch((err) => {
        this.handleError(err); // 统一的网络异常处理
        this.hideLoading(); // 关闭加载动画
        // 其他错误处理
      });
  }
}

```

输入提示功能例子

上面内容曾提到，在实际需求中会遇到这样的场景，例如当用户在一个 `<input>` 输入框中输入一些字符时，需要在输入框下方提供一些联想提示，这些提示会根据用户输入的改变而改变。这样的场景看起来很容易实现，只需要对 `<input>` 输入框进行 `change` 事件的监听，在回调函数中再把用户当前的输入内容传给服务器，接着把服务器返回的提示内容格式化后显示在输入框下方即可。这种简单的实现方式会有一些问题，比如当用户输入很快时，是否每次输入都需要请求服务器，还是在用户输入稍有停顿的时候，

才向服务器请求数据用以提示用户；再比如生产环境中会有网络延迟的问题，连续两次的 HTTP 请求往往返回的顺序会不一样，如何以正确的顺序来处理 and 显示返回的数据。

经过上述分析后，实现一个较为完善的输入提示功能，需要完成如下需求。

- 需求一：不需要每次用户输入时都发送请求到服务器上查询结果，只需确保 500ms 内用户不再输入数据时才发送请求。
- 需求二：在向服务器发送请求前检查输入内容，如果输入内容相同，则不再发送请求。
- 需求三：保证请求的响应数据不会被乱序处理。

而要实现这样的输入提示功能的解决方案想必会比较烦琐，需要维护不少状态量（用于事件回调时作为判断的依据），同时也需要实现简单的延迟触发机制等。用这种常规方式实现的代码往往难以维护，如果使用 RxJS 就可以很优雅地实现以上需求。

首先，对 `<input>` 输入框进行事件监听。示例代码如下：

```
// ...
@Component({
  selector: 'demo-input',
  template: `
    <input type="text" [ngFormControl]="term"/>
    <ul>
      <li *ngFor="let recommend of recommends">{{recommend}}</li>
    </ul>
  `,
  providers: [DemoService]
})
export class DemoInputComponent implements OnInit {
  recommends: Array<string>;
  term = new Control();

  constructor(private _demoService: DemoService) {}

  ngOnInit() {
    this.term.valueChanges.subscribe(term =>
      this._demoService.getRecommend(term).subscribe(data => {
        this.recommends = data;
      }));
  }
}
```

可以看到，在上面的代码中，最核心的部分是在 `ngOnInit()` 方法中，`this.term.valueChanges` 是一个 `Observable` 对象，这个 `Observable` 对象会在每次 `<input>` 的输入值发生改变时都抛出 `value` 值，并被 `subscribe()` 所订阅。当监听到这个 `value` 值变化时，会直接调用 `_demoService` 中的 `getRecommnd()` 方法，返回所需要的推荐数据。

现在对上述关键代码做些改造来完成之前的需求：

```
// ...
ngOnInit() {
  this.term.valueChanges
    .debounceTime(500) // 需求一：延迟 500ms
    .distinctUntilChanged() // 需求二：输入值没有变化，不需要发请求
    .switchMap(term => this._demoService.getRecommend(term)) // 需求三：保证请求
      顺序
    .subscribe(items => this.items = items);
}
```

可以看到，通过在 `this.term.valueChanges` 这个 `Observable` 对象和 `subscribe()` 之间添加的 `debounceTime()`、`distinctUntilChanged()`、`switchMap()` 三个 RxJS 内置 Operator，就可以很轻松地完成上述三个需求。

其中，`debounceTime()` 方法会过滤掉所有 `Observable` 对象抛出的时间间隔不超过 500ms 的事件，而 `disctictUntilChanged()` 方法会过滤掉所有 `Observable` 对象连续抛出的 `value` 值相同的事件，这两个 Operator 满足了需求一和需求二。

至于 `switchMap()`，这个 Operator 接收了另外一个 `Observable` 对象，即 `this._demoService.getRecommend(term)`，并把每一个 `valueChanges` 抛出来的事件映射成一个新的 `Observable` 对象，再把这些新的单一的 `Observable` 对象合并整理成一个新的 `Observable` 对象。当上游的数据流有新的数据变更时，`switchMap()` 参数里生成的新数据流会被截断，不会再往下游传递数据。如上例所示，`switchMap()` 参数是一个 AJAX 请求，当上游的输入框有新的数据输入，数据流动到 `switchMap()` 后，会截断原来 AJAX 请求的数据流，因此就不用担心 AJAX 请求返回的过时数据往下游流动，保证了最终输出的结果始终能跟输入框的值保持一致，从而优雅地实现了需求三。

当然，在实际应用中，Angular 与 RxJS 结合的场景数不胜数，这里只是给出了一小部分示例，旨在抛砖引玉，希望读者可以通过这些例子举一反三，根据自己的业务场景实现对应的需求。

9.5 小结

在本章中，首先学习了服务的概念、服务的优点，以及如何创建并使用服务；然后学习了 Angular 中最常用的服务之一，HTTP 服务；接下来通过通讯录例子讲解了如何使用 HTTP 服务向服务器拉取和发送数据，以及使用 Observable 和 Promise 两种方式分别实现 HTTP 请求的差别；最后介绍了发起跨域请求的一个解决方案，即 JSONP，并展示了在 Angular 中如何发起一个 JSONP 请求。

此外，还介绍了响应式编程的概念，RxJS 和它的 JavaScript 语言版本的实现，即 RxJS；然后罗列了 RxJS 中一些常用的操作符；最后介绍了在 Angular 中 RxJS 的使用，通过对 HTTP 服务的改造及输入提示功能的实现，说明了 RxJS 便捷的使用方式。

通过本章的学习，相信读者已经对 Angular 服务及 Rx 相关知识点有了一个较为清晰的认识。在下一章中，我们将深入探索依赖注入相关知识点，强大的依赖注入特性使得开发者在开发大型项目时，可以相互独立而高效地开发各个功能模块，同时也给应用中模块之间的解耦带来了非凡的意义。

10

依赖注入

第 9 章介绍了服务与 RxJS 的相关知识，本章将接着探讨依赖注入（Dependency Injection），它是 Angular 实现重要功能的一种设计模式。一个大型应用的开发通常会涉及很多组件和服务，这些组件和服务之间有着错综复杂的联系，如何很好地管理它们之间的依赖关系成了一个棘手的问题，而这也正是一个框架是否强大的硬指标。

Angular 提供的依赖注入机制，可以优雅地解决上面提到的问题。在传统的开发模式中，调用者负责管理所有对象的依赖，其中的循环依赖一直是梦魇。而在依赖注入模式中，这个管理权就交给了注入器（Injector），它在应用运行时（而不是发生在编译时）负责替换依赖对象，这称为控制反转（Inversion of Control，缩写为 IoC），是依赖注入的精华所在。

本章首先会从一个简单的依赖注入示例开始，介绍什么是依赖注入，接下来重点剖析 Angular 的依赖注入，最后通过例子说明在 Angular 中如何使用依赖注入。这部分内容包括如下六点：

- 在组件中注入服务。
- 在服务中注入服务。
- 在模块中注入服务。
- 层级注入。

- 注入到派生组件。
- 限定方式的依赖注入。

最后将分析 Angular 依赖注入的四种 Provider 注册形式。

通过本章的学习，相信读者对依赖注入会有一个系统的认识。

10.1 依赖注入介绍

控制反转的概念最早由 Martin Fowler 于 2004 年提出，是针对面向对象设计不断复杂化而提出的一种设计原则，是一种利用面向对象编程法则来降低应用程序耦合的设计模式。IoC 强调的是对代码引用的控制权由调用方转移到外部容器，在运行时通过某种方式（比如反射）注入进来，实现了控制的反转，这大大降低了服务类之间的耦合度。依赖注入是最常用的一种实现 IoC 的方式，也是本章将要讲解的内容。另一种是依赖查找，读者可自行在网上了解，本章不再赘述。

在依赖注入模式中，应用组件无须关注所依赖对象的创建或初始化过程，可以认为框架已经初始化好了，开发者只管调用即可。依赖注入有利于应用程序中各模块之间的解耦，使得代码更容易维护。这种优势可能一开始体现不出来，但随着项目复杂度的增加，当各模块、组件、第三方服务等相互调用更频繁时，依赖注入的优点就体现得淋漓尽致。开发者可以专注于所依赖对象的消费，无须关注这些依赖的生产过程，这无疑将大大提升开发效率。

接下来通过一个机器人的例子来加深了解依赖注入带来的好处。示例代码如下：

// 不使用依赖注入的示例

```
export class Robot {  
  public head: Head;  
  public arms: Arms;  
  constructor() {  
    this.head = new Head();  
    this.arms = new Arms();  
  }  
  // 移动机器人  
  move() {}  
}
```

一个 Robot 类会包含 Head（头）、Arms（胳膊）和 Feet（脚）等多个组件。为了方便说明，这里假设 Robot 只包含两个组件：Head 和 Arms。那么上面的代码存在什么问题呢？Robot 在它自身的构造函数中创建并引用了 Head 和 Arms 的实例，仔细分析就会发现，这种做法使得 Robot 类的扩展性差且难以测试，具体说明如下。

- 扩展性差

Robot 类通过 Head 和 Arms 创建了自己需要的组件，即头部和胳膊。试想一下，如果 Head 类的构造函数需要一个参数呢？此时没其他办法，只能通过 `this.head = new Head(theNewParameter)` 的方式修改 Robot 类；或者如果需要给 Robot 类换一个带无线通信功能的头（HeadWithWireless），只能将 Robot 引用的 Head 修改为 HeadWithWireless。

- 难以测试

当需要测试 Robot 类时，需要考虑 Robot 类隐藏的其他依赖。比如 Head 组件本身是否依赖其他组件，且它依赖的组件是否也依赖另一个组件；另外，Head 组件的实例是否发送了异步请求到服务器等。正是因为不能控制 Robot 的隐藏依赖，所以 Robot 很难被测试。

怎样才能使 Robot 类变得易于扩展且易于测试呢？这里用依赖注入的设计模式改造一下 Robot 的构造函数。示例代码如下：

```
// 使用依赖注入的示例
export class Robot {
  public head: Head;
  public arms: Arms;
  constructor(public head: Head, public arms: Arms) {
    this.head = head;
    this.arms = arms;
  }

  // 移动机器人
  move() {}
}
```

这里把依赖对象作为参数传给构造函数，在 Robot 类中不再创建 Head 和 Arms。当创建 Robot 实例时，只需把创建好的 Head 和 Arms 实例传给它的构造函数就可以了。示例代码如下：

```
var robot = new Robot(new Head(), new Arms());
```

到这一步，就实现了 Robot 类与 Head 类及 Arms 类的解耦，开发者可以将任何 Head 和 Arms 实例注入到 Robot 类的构造函数中，它们之间的注入关系如图 10-1 所示。

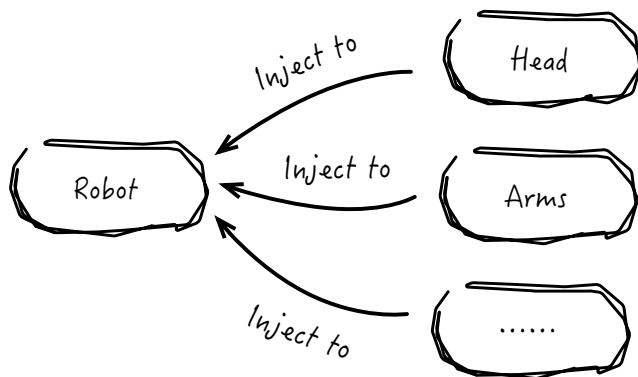


图 10-1 通过注入的方式将 Head 和 Arms 实例传给 Robot

依赖注入的一个典型应用场景就是测试，使用依赖注入方式编写的代码，测试人员在做场景覆盖测试时，基本上不需要修改被测试程序，只需要将依赖对象注入到被测试程序中即可。这里以测试 Robot 组件为例，将 Head 和 Arms 的 mock 对象传入 Robot 类的构造函数中。示例代码如下：

```
class MockHead extends Head {  
    head = '头部';  
}  
class MockArms extends Arms {  
    arms = '胳膊';  
}  
  
var robot = new Robot(new MockHead(), new MockArms());
```

依赖注入通过注入服务的方式替代了在组件里初始化所依赖的对象，从而避免了组件之间的紧耦合。但是这还不够，在使用 Robot 类时还需要手动创建 Head、Arms 及 Robot 的实例，为了减少重复操作，这里可以通过创建一个 Robot 的工厂类来解决。示例代码如下：

```
// Robot 工厂类，这并不是依赖注入的做法，仅作为引出后面内容的示例  
import { Head, Arms, Robot } from './robot';  
  
export class RobotFactory {  
    createRobot() {
```

```
    let robot = new Robot(this.createHead(), this.createArms());  
    return robot;  
}  
  
createHead() {  
    return new Head();  
}  
  
createArms() {  
    return new Arms();  
}  
}
```

上述代码只有三个方法，比较好维护，但是随着代码量的增加，维护这些代码就会变得很棘手。幸运的是，Angular 的依赖注入框架（Dependency Injection Framework）替开发者解决了这个问题。有了它，开发者就不用去关心需要定义哪些依赖，以及把这些依赖注入给谁了。因为依赖注入提供了注入器（Injector），它会帮助开发者创建所需要的类实例。例如要创建一个 Robot 类的实例，示例代码如下：

```
var injector = new Injector();  
var robot = injector.get(Robot);
```

有了注入器，Robot 就不需要知道如何创建它所依赖的 Head 和 Arms 了，用户也不需要知道如何生产一个 Robot，同时也不需要维护一个巨大的工厂类了。

通过 Robot 例子的学习，读者对依赖注入应该有了一个基本的认识——依赖注入有利于各种组件之间的解耦，代码易于维护，提升了开发效率。接下来将讨论 Angular 中的依赖注入，看它是如何使用的。

10.2 Angular 依赖注入

10.2.1 概述

在上一节中，通过 Robot 的例子讲解了依赖注入的概念，接下来将分析 Angular 的依赖注入。为了更好地理解 Angular 的依赖注入，首先介绍三个重要概念。

- 注入器（Injector）：就像制造工厂，提供了一系列的接口用于创建依赖对象的实例。

- **Provider**：用于配置注入器，注入器通过它来创建被依赖对象的实例。Provider 把标识（Token）映射到工厂方法，被依赖的对象就是通过该方法来创建的。
- **依赖（Dependence）**：指定了被依赖对象的类型，注入器会根据此类型创建对应的对象。



标识是 Angular 中 Provider 的重要概念，将在本章后面进行讲解。

如图 10-2 所示，在依赖注入中，注入器是黏合剂，它连接着调用方和被依赖方。注入器根据 Provider 的配置来生成依赖对象，调用方根据 Provider 提供的标识告诉注入器来获取被依赖的对象。

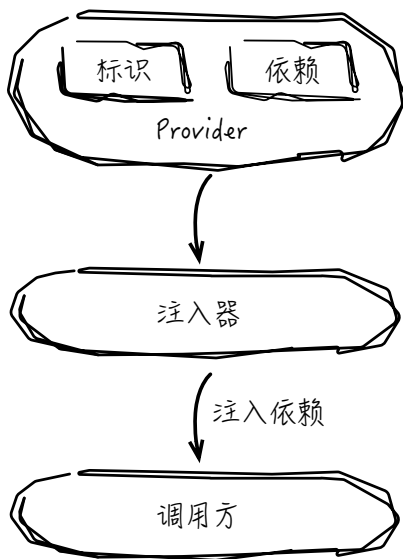


图 10-2 依赖注入组成

下面通过示例来说明在 Angular 中如何使用依赖注入获得一个 Robot 实例。示例代码如下：

```
var injector = new Injector(...)
var robot = injector.get(Robot);
robot.move(); // 调用 Robot 类的移动机器人方法
```

Injector() 的实现如下，该类暴露了一些静态方法用来创建 injector 注入器。

```
import { Injector } from '@angular/core';
```

```
var injector = Injector.create([
  [{ provide: Robot, deps: [Head, Arms] }],
  [{ provide: Head, deps: [] }],
  [{ provide: Arms, deps: [] }]]);
```

create() 是一个工厂函数, 它通过接收 Provider 数组来创建 injector 注入器, Provider 数组说明了如何创建这些依赖。事实上, 上面的 Provider 数组相当于以下方式的简写:

```
var injector = Injector.create([
  { provide: Robot, useClass: Robot, deps: [Head, Arms] },
  { provide: Head, useClass: Head, deps: [] },
  { provide: Arms, useClass: Arms, deps: [] }]);
```

Provider 对象字面量 (如 { provide: Head, useClass: Head, deps: [] }) 把一个标识映射到一个可配置的对象, 这个标识可以是一个类名, 也可以是一个字符串。有了 Provider, Angular 不仅知道使用了哪些依赖, 也知道了这些依赖是如何被创建的, 后面的内容将对 Provider 的注册方式做详细说明。

deps 显式指定了类的依赖项, 使用 Injector 时依赖需要显式指定。Injector 是从 Angular 5.0 开始加入的, 在 5.0 以前, 使用的是 ReflectiveInjector, 该类依赖 Reflect 工具, 可以自动寻找隐藏的依赖。但这个特性会逐渐被废弃, 移除对 Reflect 工具的依赖, 减少应用包体的大小。

在 Angular 中依赖 Injector 来创建注入器对象主要有三个地方, 分别是 Platform、Compiler 和 NgZone。以 Platform 为例, 在如下的启动代码中:

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

假如需要添加平台依赖模块 Robot, 且 Robot 依赖 Head 和 Arms。示例代码如下:

```
platformBrowserDynamic([
  { provide: Robot, useClass: Robot, deps: [ Head, Arms ] },
  Head,
  Arms
]).bootstrapModule(AppModule);
```

但是, 如果开发者是使用装饰器来创建模块或组件等的, 则不需要手动指定依赖 (即 deps 配置项)。示例代码如下:

```
@NgModule({  
  providers: [Robot, Head, Arms],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```



通过装饰器创建的模块或组件，其依赖实际还是借助于 Reflect 类库寻找的，但是经过 AoT 编译后依赖数据已经生成好，所以在这种情况下应用代码也不需要引入 Reflect。而自 Angular CLI 1.5.0 后，在开发阶段也可以使用 `ng serve --aot` 在 AoT 模式下开发调试，这样便完全移除了 Reflect 依赖。

10.2.2 在组件中注入服务

Angular 在底层做了大量的初始化工作，这大大简化了创建依赖注入的过程。在组件中使用依赖注入需要完成以下三个步骤：

(1) 通过 `import` 导入被依赖对象的服务。

(2) 在组件中配置注入器。在启动组件时，Angular 会读取 `@Component` 装饰器里的 `providers` 元数据，它是一个数组，配置了该组件需要使用到的所有依赖，Angular 的依赖注入框架会根据这个列表来创建对应对象的实例。

(3) 在组件构造函数中声明需要注入的依赖。注入器会根据构造函数中的声明，在组件初始化时通过第 2 步中的 `providers` 元数据配置依赖，为构造函数提供对应的依赖服务，最终完成注入的过程。

下面通过通讯录例子展示如何实现在组件中注入服务。示例代码如下：

```
// app.component.ts  
import { Component } from '@angular/core';  
  
// 1. 导入被依赖对象的服务  
import { ContactService } from '../shared/contact.service';  
import { LoggerService } from '../shared/logger.service';  
import { UserService } from '../shared/user.service';  
  
@Component({  
  selector: 'contact-app',
```

```
// 2. 在组件中配置注入器
providers: [ContactService, UserService, LoggerService],
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
}))

export class ContactAppComponent {
  // 3. 在组件构造函数中声明需要注入的依赖
  constructor(
    logger: LoggerService,
    contactService: ContactService,
    userService: UserService
  ) { }
}
```

在上述示例中，在 `ContactAppComponent` 这个根组件中配置了 `providers` 元数据，这使得 `ContactAppComponent` 及其所有子组件都能共享由根组件注入器创建的实例。需要注意的是，每个组件都可以有自己的注入器，通过依赖注入到该组件中的每个服务都维持单例。如果某个组件不希望复用从根组件注入器获取的服务，则可以在自己的注入器中以新的配置重新注入，这是 Angular 依赖注入的另一个特性，即层级注入（本章后面会详细介绍）。在下面的例子中，`CollecitonComponent` 是 `ContactAppComponent` 的子组件，它并没有在 `@Component` 中添加 `providers` 元数据来注入 `ContactService` 服务，但是依然可以在构造函数中获取到 `ContactService` 服务的实例。示例代码如下：

```
// collection.component.ts
import { Component, OnInit } from '@angular/core';
import { ContactService } from '../shared/contact.service';

@Component({
  selector: 'call-record',
  templateUrl: 'app/collection/collection.component.html',
  styleUrls: ['app/collection/collection.component.css']
})
export class CollecitonComponent implements OnInit {
  collections:any = [];
  contacts:any = {};

  constructor(private _contactService: ContactService) { }
}
```




ContactService 更像是整个模块内通用的服务，适合在全局注入，整个模块共享一个实例即可，无须在具体组件中创建相应的实例。

10.2.3 在服务中注入服务

除了组件依赖服务，服务间的相互调用也很常见。例如在上一节的 ContactService 服务中，如果希望在服务异常时记录错误信息，则可以创建一个单独的日志服务来处理。示例代码如下：

```
import { Injectable } from '@angular/core';

@Injectable()
export class LoggerService {
  log(message: string) {
    console.log(message); // 为了简单起见，这里通过控制台输出日志
  }
}
```

上面是一个简单的日志服务，在 ContactService 服务中注入 LoggerService。示例代码如下：

```
// contact.service.ts
import { Injectable } from '@angular/core';
import { LoggerService } from './logger.service';
import { UserService } from './user.service';

@Injectable() // 1. 添加装饰器 '@Injectable()'
export class ContactService {
  // 2. 在构造函数中注入所依赖的服务
  constructor(_logger: LoggerService, _userService: UserService) {}
  getCollections() {
    this._logger.log('Getting contacts...');
    //...
  }
}
```

接下来，需要在组件中注册这个日志服务。日志服务可能会被多个模块调用，建议在根模块的 providers 元数据中注册它。示例代码如下：

```
// ...  
// 3. 在组件的 providers 元数据中注册服务  
providers: [ContactService, LoggerService, UserService]  
// ...
```

注意，在 `LoggerService` 和 `ContactService` 这两个服务中都使用了 `@Injectable()` 装饰器，它到底有什么用呢？事实上它并不是必需的，只有当一个服务依赖其他服务的时候，才需要用 `@Injectable` 来显式装饰。例如上面的 `LoggerService` 服务没有依赖其他服务，它可以不用 `@Injectable` 装饰，而通讯录例子中的 `ContactService` 服务依赖了其他服务，所以这里的 `@Injectable()` 则是必需的。



Angular 官方推荐一个服务无论是否依赖其他服务，都应该使用 `@Injectable()` 来装饰服务。一方面，开发者在给某个服务注入其他服务时，无须再确认该服务是否添加了 `@Injectable()`；另一方面，这也是一种良好的团队协作方式，整个团队遵循相同的开发原则。

10.2.4 在模块中注入服务

前面提到过，在根组件中注入的服务，在所有的子组件中都能共享这个服务，当然在模块中注入服务也可以达到这样的效果。

在模块中注入服务和之前的注入场景稍有不同。Angular 在启动程序时会启动一个根模块，并加载它所依赖的其他模块，此时会生成一个全局的根注入器，由该注入器创建的依赖注入对象在整个应用程序级别可见，并共享一个实例。同时，根模块会指定一个根组件并启动，由该根组件添加的依赖注入对象在组件树级别可见，在根组件及子组件中共享一个实例。更多的关于共享实例的内容可参考下一节“层级注入”，下面先来看看在模块中添加依赖注入的例子。示例代码如下：

```
// app.module.ts  
import { NgModule } from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';  
import { AppComponent } from './app.component';  
import { LoggerService } from './shared/logger.service';  
import { UserService } from './shared/user.service';  
  
// 假设有个单独的通讯录模块  
import { ContactModule } from './contact.module';
```

```
@NgModule({
  imports: [
    BrowserModule,
    ContactModule
  ],
  declarations: [AppComponent],
  providers: [LoggerService, UserService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

在上面的代码中，通讯录功能作为独立的功能模块被封装到 `ContactModule` 模块中并通过 `import` 导入，这更符合 Angular 倡导的模块化开发，实现了模块间的松耦合。接着在 `providers` 元数据中注册了 `LoggerService`、`UserService` 这两个服务，这样在应用程序级别的其他模块都可共用这两个服务。

需要注意的是，Angular 中没有模块级作用域这个概念，只有应用程序级作用域和组件级作用域，这种设计主要考虑的是模块的扩展性，一个应用程序通常由多个模块合并而成，在 `@NgModule` 里注册的服务默认就可可在整个应用程序内可用。



延迟加载的模块是个例外，后面的路由章节会对延迟加载进行详细介绍，这里不再赘述。模块的延迟加载使得应用程序在启动时不被载入，而是结合路由配置，在需要时才动态加载相应的模块。Angular 会对延迟加载模块初始化一个新的执行上下文，并创建一个新的注入器，在该注入器中注入的依赖只在该模块内部可见，这算是模块级作用域的一个特例。

如果在多个模块中都注入了相同标识的服务怎么办？假设在根模块中先后导入了 `ContactModule` 和 `MsgModule` 两个模块。示例代码如下：

```
// ...
@NgModule({
  import: ['ContactModule', 'MsgModule']
// ...
})
// ...
```

在 `ContactModule` 和 `MsgModule` 模块中都注入了相同 Token 标识的服务。因为根注入器只有一个，后面初始化的模块服务会覆盖前面初始化的模块服务，如上例 `MsgModule` 中初始化的服务会覆盖 `ContactModule` 中初始化的服务，而且即便是 `ContactModule` 模块里的组件，且这些组件引入的是同一个 Token 标识的服务，那么这些组件所引入的服务也依然会是 `MsgModule` 模块里注入的那个服务实例，这种情况需要特别注意。

还有一种场景是，假如是 `ContactModule` 导入 `MsgModule` 的。示例代码如下：

```
// ...
@NgModule({
  Imports: [ MsgModule ],
  // ...
})
export class ContactModule { }
```

在这种情况下，应用程序里使用的服务会是 `ContactModule` 中的注入服务，而不是 `MsgModule` 里的。按照这种结论延伸，在根模块里注入的服务始终都是有最高优先级的，所以可以放心使用根模块里注入的服务。

建议在根模块中集中管理其他模块的导入，通过 `providers` 元数据完成配置。另外，可以利用模块延迟加载的特性，在延迟模块中注入依赖，或者在模块的根组件中注入依赖。以上方法都可以避免多模块的相同标识污染的问题。

那么，服务是在模块中注入还是在根组件中注入，该怎么选择呢？这主要取决于该服务的应用场景，在模块中注入的服务的作用域是应用程序级别的，像日志等工具类服务可能会在多个模块中调用，所以更适合在模块中注入；而类似于 `ContactService`、`MsgService` 等与业务场景相关的服务，可在相关模块的根组件中注入。由于存在延迟加载模块调用不到组件级别作用域里服务的情况，如果一个服务需要被延迟加载的模块调用，则也应该在根模块中注入。最后，如果不确定一个服务将来是否会被外部模块调用，则可优先考虑在模块中注册。

10.2.5 层级注入

在上面的内容中，介绍了 `Angular` 中依赖注入的几个场景。`Angular` 以组件为基础，在项目开发中组件自然会有层层嵌套的情况，这种组织关系组成了组件树（如图 10-3 所示）。在根组件下面是各层级的子组件，被注入的依赖对象就像每棵树上结的果实，可以出现在任何层级的任何组件中，每个组件都可以拥有一个或多个依赖对象的注入（一个或多个果实），对于注入器而言每个依赖对象都是单例（`Singleton`）。

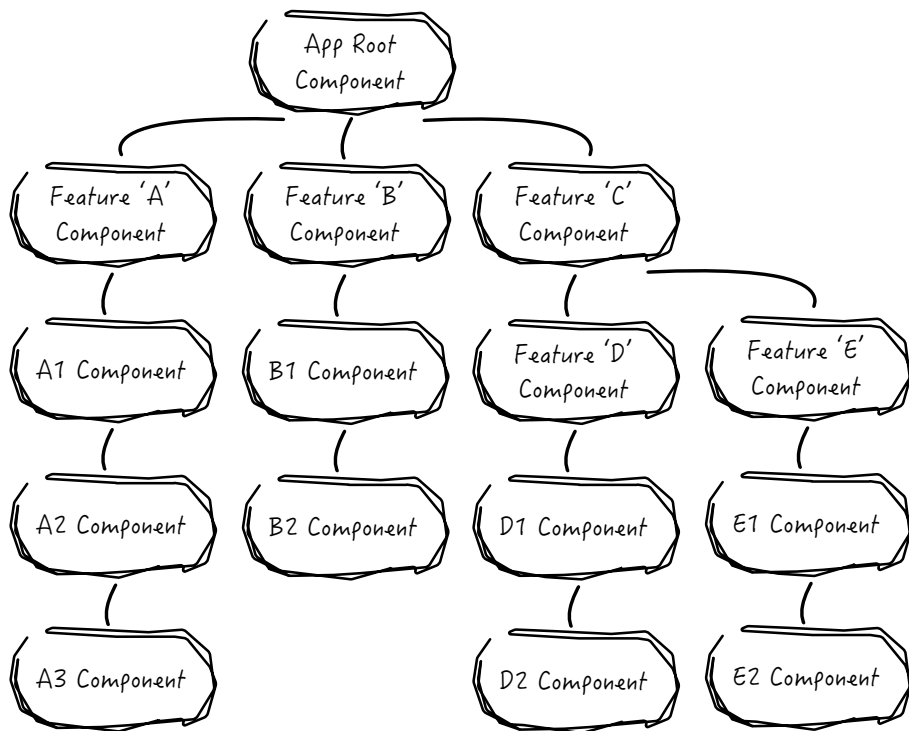


图 10-3 组件树



实际上，每个组件都有自己的注入器（但并不是每个组件都会为自己创建独立的注入器，也有可能是共享其他组件的注入器），由这个注入器创建的 Provider 都是单例，这是组件级别的单例，跟 AngularJS 1.x 不一样，有 AngularJS 1.x 开发经验的读者会更有体会，AngularJS 1.x 只能是全局单例。

上面提到注入可以发生在整棵组件树的任一层级，并在各层级组件的注入器中维持单例。更进一步来说，依赖注入可以传递到子孙组件中，子组件可以共享父组件中注入的实例，无须再创建。如果注入了多个实例，这些后代（子孙）组件也都可以共享这些实例，如图 10-4 所示。

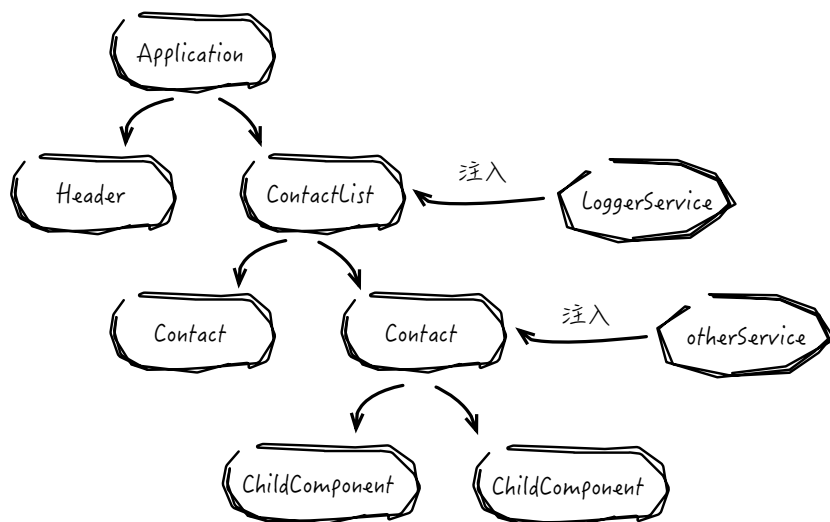


图 10-4 层级注入

在通讯录例子中，假设通讯录列表需要给每个通讯录子项生成一个唯一标识，则可以通过在子组件中注入随机数服务来实现。示例代码如下：

```
// ...
// 生成唯一标识的服务
@Injectable()
export class Random {
  constructor() {
    this.num = Math.random();
  }
}

// ...
// 子组件 A
@Component({
  selector: 'contact-a',
  providers: [Random], // 单例
  template: '<div>ContactA : {{ random.num }}</div>'
})
export class ContactAComponent {
  constructor(r: Random){
    this.random = r;
  }
}
```

```
// ...
// 子组件 B
@Component({
  selector: 'contact-b',
  providers: [Random], // 单例
  template: '<div>ContactB : {{ random.num }}</div>'
})
export class ContactBComponent {
  constructor(r: Random) {
    this.random = r;
  }
}

// ...
// 父组件
@Component({
  selector: 'contact-list',
  template: `
    <h1>Contact-list</h1>
    <contact-a></contact-a>
    <contact-b></contact-b>
  `,
})
export class ContactListComponent {
  constructor() {}
}
```

结果将输出：

ContactA : 0.6594064461255684

ContactB : 0.9344545328845055

上述输出结果说明每个子组件都创建了自己独立的注入器，也就是说，通过依赖注入的 Random 服务都是独立的。如果把组件的 providers 元数据配置稍微改一下，把注入器提升到父组件中，那么输出的结果将会不一样。示例代码如下：

```
// 子组件 A
// 在组件元数据中没有单独指定 providers，共享父组件的注入器
@Component({
  selector: 'contact-a',
  template: '<div>ContactA : {{ random.num }}</div>',
})
```

```
// 子组件 B
// 在组件元数据中没有单独指定 providers，共享父组件的注入器
@Component({
  selector: 'contact-b',
  template: '<div>ContactB : {{ random.num }}</div>'
})

// 父组件
@Component({
  selector: 'contact-list',
  template: `
    <h1>contact-list</h1>
    <contact-a></contact-a>
    <contact-b></contact-b>
  `,
  providers: [Random] // 指定 providers，创建独立的注入器
})
```

此时输出结果变成：

```
ContactA : 0.027503783541033888
ContactB : 0.027503783541033888
```

上述输出结果说明子组件继承了父组件的注入器，所以子组件使用了相同的 `Random` 实例，输出相同的结果，也就不能满足通讯录子项需要有唯一标识的需求了。

Angular 这种灵活的设计引出了这样一个思考：是在根组件还是在子组件中进行服务注入，该怎么选择呢？这取决于想让注入的依赖服务具有全局性还是局部性。由于每个注入器总是将它提供的服务维持单例，因此，如果不需要针对每个组件都提供独立的服务单例，就可以在根组件中注入，整个组件树共享根注入器提供的服务实例，如在上述一些组件中使用的日志工具类等；如果需要针对每个组件创建不同的服务实例，就应该在各子组件中配置 `providers` 元数据来注入服务。

另一个问题是，Angular 是如何查找到合适的服务实例的呢？当组件的构造函数试图注入某个服务的时候，Angular 会先从当前组件的注入器查找，找不到就继续到父组件的注入器查找，直到根组件注入器，最后到应用根注入器（即模块注入器），此时找不到的话就报错，图 10-5 展示了组件往上查找服务的过程。当然，后面内容介绍的限定依赖注入是个例外，它可以控制查找的范围，即使找不到也不报错。

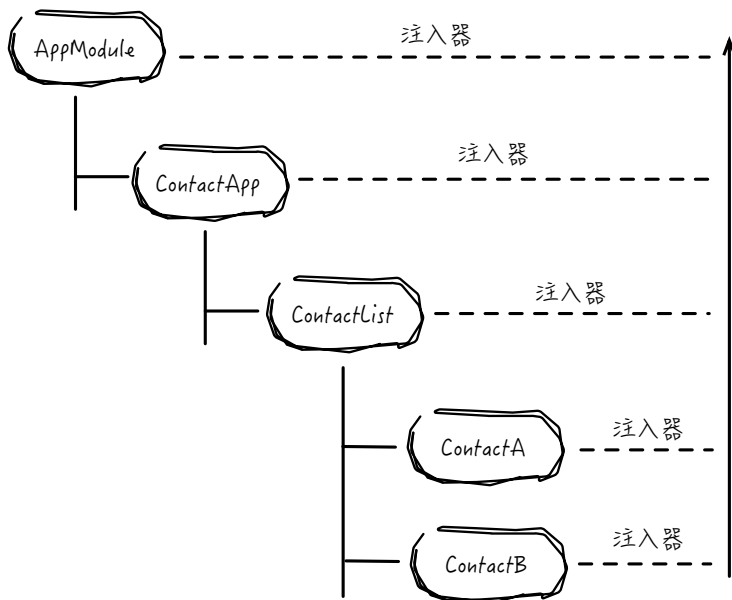


图 10-5 注入服务的查找路径

10.2.6 注入到派生组件

一个组件可以派生（Inherit）于另一个组件，对于有继承关系的组件，当父类组件和派生类组件有相同的依赖注入时，如果父类组件注入了这些依赖，那么派生类组件也需要注入相同的依赖，并在派生类组件的构造函数中通过 `super()` 往上传递。



组件本质上是一个类，而类有继承的关系，所以一个组件可以继承另一个组件。但派生类组件不能继承父类组件的注入器，二者的注入器对象并没有任何关联，而且需要注意的一点是，因为父类组件的运行可能需要依赖注入某些服务，所以派生类组件也必须注入父类组件依赖的服务，然后调用 `super()` 将对应的注入服务传递到父类。

关于父组件跟子组件、父类组件跟派生类组件的称谓，前者是聚合关系，后者是继承关系。

在通讯录例子中，假设需要对返回的通讯录列表排序，则可以创建一个实现排序功能的派生组件。示例代码如下：

```
// 父类组件示例代码  
// ...
```

```
@Component({
  selector: 'contact-app',
  providers: [ContactService],
  templateUrl: 'app/contact-app.html'
})

export class ContactAppComponent implements OnInit {
  collections: any = {};
  constructor(protected _contactService: ContactService) {}
  ngOnInit() {
    this._contactService.getCollections().subscribe(data =>{
      this.collections = data;
      this.afterGetContacts();
    });
  }
  protected afterGetHeroes() {}
}
```

实现排序功能的派生组件。示例代码如下：

```
// ..
@Component({
  selector: 'contact-app',
  // 在派生组件中注入
  providers: [ContactService],
  templateUrl: 'app/contact-app.html'
})

// 继承父类组件
export class SortedContactAppComponent implements ContactAppComponent {
  // 在派生组件中注入
  constructor(protected _contactService: ContactService) {
    // 往父类组件传递
    super(_contactService);
  }

  ngOnInit() {}

  protected afterGetContacts() {
    this.collections = this.collections.sort((h1, h2) =>{
```

```
        return h1.name < h2.name ? -1 : (h1.name > h2.name ? 1 : 0);
    });
}
}
```

上面的例子基于之前的通讯录例子做了修改，将获取通讯录列表信息的代码移到了 `ngOnInit()` 里，同时重写了 `afterGetContacts()` 这个方法。在获取数据后进行排序，派生组件重写了排序的具体实现逻辑，最终派生类的 `collections` 就是排序后的结果。派生类组件的 `providers` 注入步骤在上述示例中是必需的，因为注入器不能从父类继承。如果把 `providers` 配置放到模块中，那么上述例子中父类或派生类都不需要配置 `ContactService` 这个服务，因为它们共享同一个 `ContactService` 实例。



Angular 推荐的最佳实践是构造函数越简单越好，尽量只负责类似于变量初始化这样的操作，业务逻辑应移到 `ngOnInit()` 中处理，这有利于组件的单元测试。

10.2.7 限定方式的依赖注入

到目前为止，注入都是假定依赖对象是存在的，然而实际情况往往并非如此，比如上层提供的 `Provider` 被移除，导致之前注入的依赖可能已经不存在了，此时再按照前面讲的依赖注入方式进行相关服务的调用，应用就会出错。幸运的是，Angular 依赖注入框架提供了 `@Optional` 和 `@Host` 装饰器来解决上面提到的问题。

Angular 的限定注入方式使得开发者能够修改默认的依赖查找规则，`@Optional` 可以兼容依赖不存在的情况，提高系统的健壮性。`@Host` 可以限定查找规则，明确实例初始化的位置，避免一些莫名的共享对象问题。

在 Angular 中实现可选注入很简单，在宿主组件（Host Component）的构造函数中增加 `@Optional()` 装饰器即可。示例代码如下：

```
import { Optional } from '@angular/core';
import { LoggerService } from '../shared/logger.service';

constructor(@Optional() private logger: LoggerService) {
    if (this.logger) {
        this.logger.log('hello');
    }
}
```

这样就能兼容 `LoggerService` 服务不存在的场景了。



事实上, `LoggerService` 这个类的定义还是“存在”的, 上述“不存在”是指这个类虽然定义了, 但并未“准备好”, 没有在相应的组件或模块中通过 `providers` 元数据来配置它。

另外, 依赖查找的规则是按照注入器从当前组件向父级组件查找的, 直到找到要注入的依赖为止。但有时候想限制这种默认的查找规则, 比如限定查找的路径截止在宿主组件, 如果找不到就会出错, 而不是继续往上查找, `Angular` 提供的 `@Host` 装饰器可以用于解决这个问题。

如果一个组件注入了依赖项, 那么该组件就是这个依赖项的宿主组件。但如果这个组件通过 `ng-content` 被嵌入到父组件中, 那么这个父组件就是该依赖项的宿主组件。下面会分别举例说明这两种情形。

宿主组件是当前组件

在通讯录例子中, 因为 `LoggerService` 服务已经在顶层组件里通过 `providers` 元数据配置了, 根据查找规则, 最终会在顶层组件里查找到依赖, 代码可以正常运行。示例代码如下:

```
// ...
@Component({
  selector: 'contact-list',
  template: `
    <h1>Contact List</h1>
    <contact-a></contact-a>
  `
})
export class ContactListComponent {
  constructor(logger: LoggerService) {}
}
```

如果加入 `@Host` 装饰器来限定查找规则只停止于当前组件, 则 `Angular` 就会抛出错误。当然, 可以结合 `@Optional` 装饰器来跳过这种检查。示例代码如下:

```
// ...
@Component({
  selector: 'contact-list',
```

```

    template: `
      <h1>Contact List</h1>
      <contact-a></contact-a>
    `,
  })
}
export class ContactListComponent {
  constructor(
    @Host() // 错误！找不到 LoggerService 实例，因为实例的查找限制在宿主组件中
    logger: LoggerService) {}
}

```

结合 @Optional 使得检查通过。示例代码如下：

```

// ...
@Component({
  selector: 'contact-list',
  template: `
    <h1>Contact List</h1>
    <contact-a></contact-a>
  `,
})
export class ContactListComponent {
  constructor(
    @Host()
    @Optional() // 正确！虽然 LoggerService 实例的查找只限制在宿主组件中，但是加上可选参数后不报错
    logger: LoggerService){}
}

```

宿主组件是父组件

这里修改一下父组件 ContactListComponent 和子组件 ContactAComponent 的代码，修改为在 ContactAComponent 组件中注入 LoggerService 服务，并将 ContactAComponent 通过 ng-content 的方式嵌入到父组件 ContactListComponent 中，看看会是什么情况。

在模板中使用这两个组件的示例代码如下：

```

<contact-list>
  <contact-a></contact-a>
</contact-list>

```

修改后的 ContactListComponent 和 ContactAComponent 组件定义如下：

```
// ...
// ContactListComponent 组件核心代码
@Component({
  selector: 'contact-list',
  template: `
    <h1>Contact List</h1>
    <ng-content></ng-content> // ContactAComponent 组件模板的内容将会被嵌入到这里
  `,
  providers: [LoggerService] // 在 ContactListComponent 中注入独立的 LoggerService
})
export class ContactListComponent {
  constructor() {}
}

// ...
// ContactAComponent 组件核心代码
@Component({
  selector: 'contact-a',
  template: '<div>ContactA</div>'
})
export class ContactAComponent {
  constructor(
    @Host()
    @Optional()
    logger: LoggerService
  )
}
```

如上述代码所示，因为子组件 ContactAComponent 通过 ng-content 的方式被嵌入到父组件 ContactListComponent 中，所以 ContactListComponent 就变成了 ContactAComponent 的宿主，最终在 ContactAComponent 中注入的 LoggerService 服务，会向上找到 ContactListComponent 组件配置的 LoggerService 服务。

10.3 Provider

10.3.1 概述

在上面的内容中，通过例子说明了不同形式的依赖注入，Provider 这个概念一直有意不去介绍，这一节将详细讲解。

Provider 设计模式由来已久，在前后台各种技术领域中被广泛使用，如 .Net 的 MembershipProvider、Hibernate 的 ConnectionProvider，以及 Android 的 ContentProvider 等。Provider 实现了逻辑操作或数据操作的封装，以接口的方式提供给调用方使用，Provider 模式提供了很好的可扩展性和灵活性。

在 Angular 中，Provider 描述了注入器如何初始化标识（Token）所对应的依赖服务，它最终被用于注入到组件或者其他服务中，这个过程好比厨师（注入器）根据菜谱（Provider）制作一道名为 LoggerService（标识）的菜（依赖服务）。Provider 提供了一个运行时所需的依赖，注入器依靠它来创建服务对象的实例。

在上面的日志服务 LoggerService 例子中，LoggerService 被注册到了 providers 数组（元数据）中。示例代码如下：

```
@Component({  
  // ...  
  providers: [LoggerService],  
  // ...  
})
```

实际上它的完整形式如下：

```
@Component({  
  // ...  
  providers: [{provide: LoggerService, useClass: LoggerService}]  
  // ...  
})
```

上面代码的完整形式采用了对象字面量的方式来描述一个 Provider 的构成要素。其中 provide 属性可以理解为这个 Provider 的唯一标识，用于定位依赖值，以及注册 Provider，也就是应用中使用服务名，而 useClass 属性则代表使用哪个服务类来创建实例。

Angular 的 Provider 引进标识机制解决了 AngularJS 1.x 版本中存在的几个痛点：

- 标识是字符串，作为唯一标识（Angular 会有一个标识映射表，保证唯一性），不再依赖具体的类，可避免命名空间污染。
- 代码只依赖一个抽象的标识，不再依赖具体的实现，可以在运行时动态替换。



事实上,标识可以是字符串,也可以是其他数据类型。当多个字符串同时映射到同一个标识的时候,会以最后一个为准,这可能会导致一些隐含的缺陷。为了解决标识命名冲突的问题,Angular 引入了 OpaqueToken (不透明标识),可以保证所生成的标识都是唯一的。关于详细的 OpaqueToken 用法,读者可以在官网上进一步了解。

10.3.2 Provider 注册方式

Provider 的主要作用是注册并返回合适的服务对象。Angular 提供了如下四种常见的 Provider 注册形式,开发者可以结合不同的场景来选择使用。

- 类 Provider
- 值 Provider
- 别名 Provider
- 工厂 Provider

类 Provider

类 Provider 基于标识来指定依赖项,这种方式可以使得依赖项能够被动态指定为其他不同的具体实现,只要接口不变,对于使用方就是透明的。一个典型的场景就是数据渲染服务 (Render), Render 服务对上层应用提供的接口是固定的,但底层可以用 DOM 渲染方式 (DomRender),也可以用 Canvas 渲染方式 (CanvasRender),还可以用 Angular Universal 实现服务端渲染 (ServerRender),如图 10-6 所示。

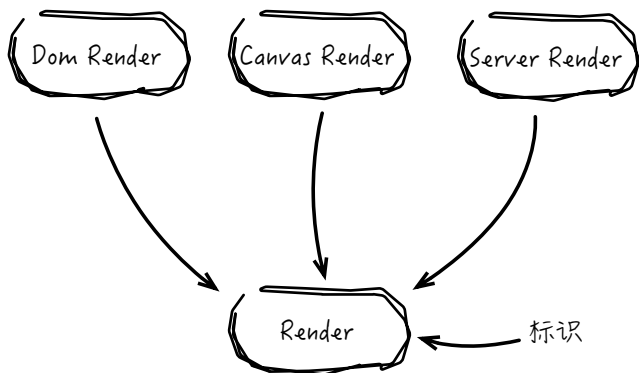


图 10-6 类 Provider 的例子

然后通过 `useClass` 属性来指定使用哪种渲染方式。因为渲染服务的最终接口并没有变化（例子中还是 `Render`），这对于调用者来说，业务代码无须修改，从而带来了极大的便利性。示例代码如下：

```
// 渲染方式
var injector = Injector.create([
  {provide: Render, useClass: DomRender, deps: []} // DOM 渲染方式
  // {provide: Render, useClass: CanvasRender, deps: []} // Canvas 渲染方式
  // {provide: Render, useClass: ServerRender, deps: []} // 服务端渲染方式
]);

// 调用方
class ApplicationComponent{
  constructor(_render: Render){
    _render.render(); // 渲染
  }
}
```

值 Provider

在实际项目中，依赖对象不一定是类，也可以是常量、字符串、对象等其他数据类型，以方便使用在全局变量、系统相关参数配置等场景中。在创建 `Provider` 对象时，只需使用 `useValue` 就可声明一个值 `Provider`。示例代码如下：

```
// ...
let globalSetting = {
  env: 'production',
  getHost: () => { return 'https://angular.io' }
};

@Component({
  selector: 'some-component',
  template: '<div>Some Component</div>',
  providers: [
    { provide: 'urlSetting', useValue: globalSetting }, // 对象
    { provide: 'NAME', useValue: '揭秘 Angular' } // 常量
  ]
})
export class SomeComponent {
  constructor() {}
}
```



值 Provider 依赖的值（通过 `useValue` 指定的值）必须在当前或者 `providers` 元数据配置之前定义。标识为 `NAME` 依赖的值在 Provider 中直接给出了定义，即定义 `useValue` 的值为“揭秘 Angular”，而标识为 `urlSetting` 依赖的值为变量 `globalSetting`，变量在使用之前需先定义好。

别名 Provider

有了别名 Provider，就可以在一个 Provider 中配置多个标识，其对应的对象指向同一个实例，从而实现多个依赖、一个对象实例的作用。`useExisting` 可以用来指定一个别名 Provider。

假如应用已有一个日志服务 `OldLoggerService`，现在开发了有相同接口的新版服务 `NewLoggerService`，考虑到重构代价等原因，并不想去替换 `OldLoggerService` 服务被使用的地方。此时为了让新旧服务同时可用，可以用 `useClass` 来解决这个问题。示例代码如下：

```
// ...
providers: [
  {provide: NewLoggerService, useClass: NewLoggerService},
  {provide: OldLoggerService, useClass: NewLoggerService}
]
// ...
```

但是，上述两个 `NewLoggerService` 是不同的实例，这显然不是我们预期的效果。幸运的是，Angular 已经考虑到这种情况，可以在创建 Provider 对象时使用 `useExisting`，使得多个标识指向同一个实例。示例代码如下：

```
// ...
providers: [
  {provide: NewLoggerService, useClass: NewLoggerService},
  {provide: OldLoggerService, useExisting: NewLoggerService}
]
// ...
```

工厂 Provider

有时候依赖对象是不明确且动态变化的，可能需要根据运行环境、执行权限来生成，Provider 需要一种动态生成依赖对象的能力。Angular 提供的工厂 Provider 可以解决

这个问题，它通过暴露一个工厂方法，返回最终依赖的对象。

在通讯录例子中，假设有这样一个场景：有些联系人的信息是保密的，只有拥有特定权限的人才能看到，所以需要每个登录用户进行鉴权。要达到这样的目的，可以在构造函数中通过一个布尔值来判断是否有权限并返回对应的服务，在返回的服务中可以根据这个布尔值来判断是否显示联系人信息。示例代码如下：

```
let contactServiceFactory = (_logger: LoggerService, _userService: UserService) =>
{
  return new contactService(_logger, _userService.user.isAuthorized);
}

export let contactServiceProvider =
{ provide: ContactService, useFactory: contactServiceFactory, deps: [
  LoggerService, UserService] };
```

使用工厂 Provider 的注册方式需要用 useFactory 来声明 Provider 是一个工厂方法，如上例中指定具体的实现方法是 contactServiceFactory(); deps 是一个数组属性，指定了所需要的依赖，可以注入到工厂方法中。

上面几种 Provider 的注册方式可根据不同的场景选择使用，为开发大型复杂的应用提供了便利。

10.4 扩展阅读

除了上面章节中提到的应用场景，依赖注入还应用在其他场景中。在“组件”章节中，已经介绍了组件间通信的几种方式，其中包括父组件获取子组件引用的方式（通过 ViewChildren 的方式）。反过来，在子组件中获取父组件的实例就相对麻烦一些，考虑到每个组件的实例都会添加到注入器的容器里，因此可通过依赖注入来找到父组件的实例。

假设有一个父组件 ParentComponent 和一个子组件 ChildComponent。示例代码如下：

```
// parent.component.ts
@Component({
  selector: 'parent',
  template: `
    <div>
      <h3>{{name}}</h3>
```

```

        <child></child>
    </div>
    、
})
export class ParentComponent {
    name = '父组件';
}

```

现在要在子组件中获取父组件的引用，有两种情况。

- 已知父组件的类型

这种情况可以直接通过在构造函数中注入 `ParentComponent` 来获取已知类型的父组件引用。示例代码如下：

```

// child.component.ts
import { ParentComponent } from './parent.component';

@Component({
    selector: 'child',
    template: `
        <div class="c">
            <h3>{{name}}</h3>
            <div>
                {{parent ? '获取到父组件实例' : '获取不到父组件实例'}}
            </div>
        </div>
    `
})
export class ChildComponent {
    name = '子组件';
    constructor(public parent: ParentComponent) { }
}

```

- 未知父组件的类型

一个组件可能是多个组件的子组件，有时候无法直接知道父组件的类型，在 `Angular` 中，可通过“类—接口（Class-Interface）”的方式来查找，即让父组件通过提供一个与“类—接口”标识同名的别名来协助查找。



“类—接口”其实是一个抽象类，但被当作接口来使用。因为接口是 TypeScript 里才有的概念，编译后并不存在，因此 Provider 的标识不能是接口，只能是 JavaScript 对象（函数、字符串、对象等），所以“类—接口”既能提供接口的强类型约束，又能当作 Provider 的标识来使用。

首先创建 Parent 抽象类，它只声明了 name 属性，没有实现（赋值）。示例代码如下：

```
export abstract class Parent {  
  name: string;  
}
```

然后在 ParentComponent 组件的 providers 元数据中定义一个别名 Provider，用 useExisting 来注入 ParentComponent 组件的实例。示例代码如下：

```
@Component({  
  selector: 'parent',  
  template: `  
    <div>  
      <h3>{{name}}</h3>  
      <child></child>  
    </div>  
  `,  
  providers: [{provide: Parent, useExisting: ParentComponent }]  
})  
export class ParentComponent implements Parent {  
  name = '父组件';  
}
```



如果还有多层组件，建议任何父组件都应该实现一个抽象类（如上面的 Parent 抽象类），这样子组件就可以通过注入找到这些父组件的实例，比如上面的 ParentComponent 组件类。

通过下面的代码，就可以在子组件中通过 Parent 这个标识找到父组件的实例了。

```
// ...  
export class ChildComponent {  
  name = '子组件';  
  constructor(public parent: Parent ) {}  
}
```

10.5 小结

在本章中，首先介绍了依赖注入的基本概念。重点剖析了 Angular 的依赖注入，分析了 Angular 依赖注入涉及的三个重要概念：注入器（Injector）、Provider 和依赖（Dependence）。通过例子说明了 Angular 中依赖注入的使用，包括在不同场景（组件、服务、模块）中注入服务的方式及层级注入等。同时也介绍了使用限定方式的依赖注入可以解决哪些问题，以及如何保证依赖的健壮性。

接下来讲解了 Angular 的四种依赖注入形式，分别是：类 Provider、值 Provider、别名 Provider 和工厂 Provider。

最后介绍了如何通过注入查找父组件实例，包括两种情况：一是已知父组件的类型，直接使用标准的依赖注入来获取父组件；二是未知父组件的类型，可以通过“类—接口”的方式来查找，即父组件通过提供一个与“类—接口”标识同名的别名来协助查找。

通过本章的学习，读者对于什么是依赖注入，以及在 Angular 中如何使用依赖注入有了一个清晰的认识。在第 11 章中，我们将探索 Angular 的路由、子路由和路由的生命周期等相关内容。

11 路由

在“第 5 章 Angular 架构总览”中讲到，一个 Angular 应用通常包含多个功能相对独立的模块，每个模块包含各自的组件，不同模块的不同组件在路由的统一管理下协同工作。

前面的章节介绍了如何开发组件和模块，本章将介绍路由的相关内容，包括路由的基本原理、路由跳转、参数传递等，以及 Angular 应用如何使用路由来对各组件进行管理，如何通过对各组件进行灵活搭配来满足不同业务场景的需求。

11.1 概述

路由所要解决的核心问题是通过建立 URL 和页面的对应关系，使得不同的页面可以用不同的 URL 来表示。主流前端框架围绕这个问题给出了各自的路由实现，虽然语法和工作机制不尽相同，但理念却殊途同归。在 Angular 中，页面由组件构成，因此 URL 和页面的对应关系实质上就是 URL 和组件的对应关系。

建立 URL 和组件的对应关系可通过路由配置来指定。如图 11-1 所示，路由配置包含了多个配置项。最简单的情况是，一个配置项包含了 `path` 和 `component` 两个属性，其中 `path` 属性将被 Angular 用来生成一个 URL，而 `component` 属性则指定了该 URL 所对应的组件。

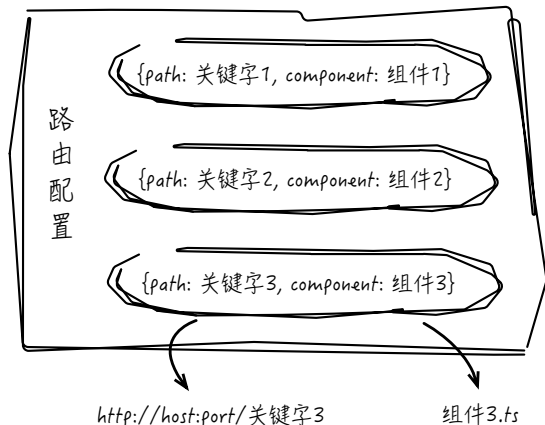


图 11-1 路由配置

在定义了路由配置后, Angular 路由将以其为依据来管理应用中的各个组件。图 11-2 展示了 Angular 路由的核心工作流程。

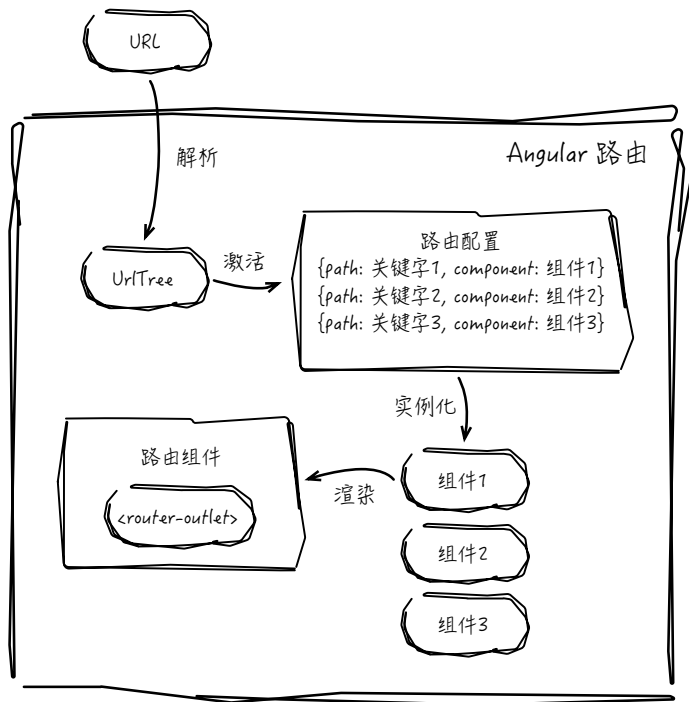


图 11-2 Angular 路由的核心工作流程

- 首先，当用户在浏览器地址栏中输入 URL 后，Angular 将获取该 URL 并将其解析生成一个 `UrlTree` 实例。
- 其次，在路由配置中寻找并激活与 `UrlTree` 实例匹配的配置项。
- 再次，为配置项中指定的组件创建实例。
- 最后，将该组件渲染于路由组件的模板中 `<router-outlet>` 指令所在的位置。

本章对于各路由功能的介绍，都将围绕此工作流程来展开。

11.2 基本用法

Angular 路由最基本的用法是将一个 URL 所对应的组件在页面中显示出来。要做到这一点，有三个必不可少的步骤，分别是定义路由配置、创建根路由模块、添加 `RouterOutlet` 指令。本节将以联系人列表页和收藏页为例，演示如何完成这几个步骤。

11.2.1 路由配置

路由配置是一个 `Routes` 类型的数组，如下面的 `rootRouterConfig` 数组所示，数组的每一个元素即为一个路由配置项。下面的代码定义了两个配置项，在默认路由策略 `PathLocationStrategy` 下（关于路由策略后面会继续展开讲解），第一个配置项中 `path` 值对应的 URL 为 `http://localhost:3000/list`，与 `ListComponent` 组件相关联；第二个配置项中 `path` 值对应的 URL 为 `http://localhost:3000/collection`，与 `CollectionComponent` 组件相关联。

```
// app.routes.ts
import { Routes } from '@angular/router';

import { ListComponent } from './list/list.component';
import { CollectionComponent } from './collection/collection.component';

export const rootRouterConfig: Routes = [
  // ...
  { path: 'list', component: ListComponent }, // http://localhost:3000/list
  { path: 'collection', component: CollectionComponent } // http://localhost:3000/
    collection
];
```

11.2.2 创建根路由模块

根路由模块包含了路由所需的各项服务，是路由工作流程得以正常执行的基础。下面的代码以路由配置 `rootRouterConfig` 为参数，通过调用 `RouterModule.forRoot()` 方法来创建根路由模块，并将其导入到应用的根模块 `AppModule` 中。

```
// app.module.ts
import { ModuleWithProviders } from '@angular/core';
import { RouterModule } from '@angular/router';

Import { rootRouterConfig } from './app.routes';

let rootRouterModule: ModuleWithProviders = RouterModule.forRoot(rootRouterConfig);

@NgModule({
  imports: [rootRouterModule],
  //...
})
export class AppModule {}
```

需要注意的是，根路由模块默认提供的路由策略为 `PathLocationStrategy`。该策略要求应用必须设置一个 `base` 路径，用于作为前缀来生成和解析 URL。设置 `base` 路径最简单的方式是在 `index.html` 文件中设置 `<base>` 元素的 `href` 属性。路由策略将会在下一节进行详细介绍。

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <base href="/">
    <!-- ... -->
  </head>
  <body>
    <!-- ... -->
  </body>
</html>
```

11.2.3 添加 RouterOutlet 指令

`RouterOutlet` 指令的作用是在组件的模板中开辟出一块区域，用于显示 URL 所对应的组件。Angular 将模板中使用了 `<router-outlet>` 标签的组件统称为路由组件。下面的代码在根组件 `AppComponent` 的模板中添加了一个 `<router-outlet>`。

```
<!-- app.component.html -->
<main class="main">
  <router-outlet></router-outlet>
</main>
```

至此，就完成了具备基本路由功能的 Angular 应用。以本书中通讯录例子的联系人列表页为例，当在浏览器地址栏中输入 `http://localhost:3000/list` 后，便可以看到 `ListComponent` 组件在页面上显示出来。所生成的 HTML 代码（片段）如下：

```
<body>
  <!-- 根组件 -->
  <contact-app>
    <main class="main">
      <router-outlet></router-outlet>
      <!-- ListComponent 组件 -->
      <list>
        <!-- ... -->
      </list>
    </main>
  </contact-app>
</body>
```

11.3 路由策略

路由策略决定 Angular 将使用 URL 的哪一部分来和路由配置项的 `path` 属性进行匹配。图 11-3 显示了 URL 中的路由策略，Angular 提供了 `PathLocationStrategy` 和 `HashLocationStrategy` 两种路由策略，分别表示使用 URL 的 `path` 部分和 `hash` 部分来进行路由匹配。

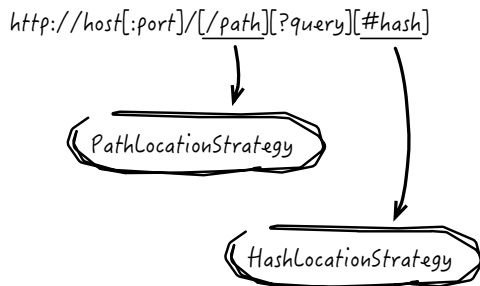


图 11-3 URL 中的路由策略

以通讯录例子的联系人列表页的配置项为例，随着所选取的路由策略的不同，与之所对应的 URL 也略有不同，如下所示：

- 路由策略为 PathLocationStrategy 时，URL 是 `http://localhost:3000/list`。
- 路由策略为 HashLocationStrategy 时，URL 是 `http://localhost:3000/##/list`。

11.3.1 HashLocationStrategy 介绍

HashLocationStrategy 是 Angular 路由最常见的策略，其原理是利用了浏览器在处理 hash 部分的两个特性。

第一，浏览器向服务器发送请求时不会带上 hash 部分的内容。如图 11-4 所示，如果通讯录采用了 HashLocationStrategy，那么对于其所有配置项所对应的 URL，浏览器向服务器发送的请求都为同一个，服务器只需要返回应用首页即可，Angular 在获取首页后会根据 hash 的内容来匹配路由配置项并渲染相应的组件。

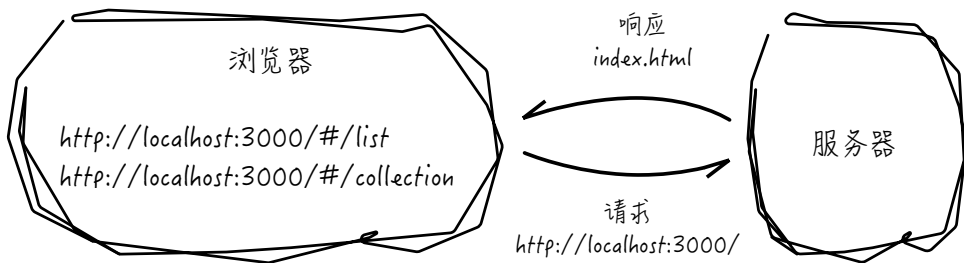


图 11-4 HashLocationStrategy 请求处理

第二，更改 URL 的 hash 部分不会向服务器重新发送请求，这使得在进行跳转的时候不会引发页面的刷新和应用的重新加载。

要使用该策略，只需要在注入路由服务时使用 `useHash` 属性进行显式指定即可。

```
// app.module.ts
@NgModule({
  imports: [RouterModule.forRoot(rootRouterConfig, {useHash: true})],
  // ...
})
export class AppModule {}
```

11.3.2 PathLocationStrategy 介绍

PathLocationStrategy 使用 URL 的 path 部分来进行路由匹配，因此与 HashLocationStrategy 的不同之处在于，浏览器会将配置项对应的 URL 原封不动地发送给服务器，如图 11-5 所示。

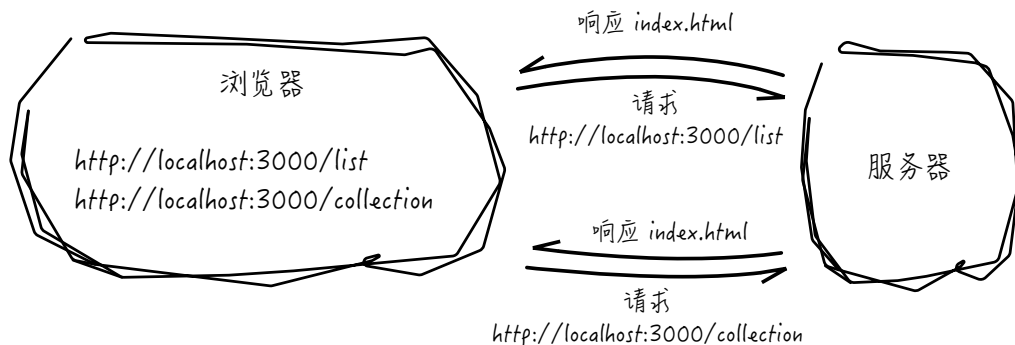


图 11-5 PathLocationStrategy 请求处理

作为 Angular 的默认路由策略，其最大的优点在于为服务器端渲染提供了可能。比如，当使用 PathLocationStrategy 策略获取联系人列表页时，浏览器会向服务器发送请求 `http://localhost:3000/list`，服务器可以通过解析 URL 的 path 部分 `/list` 得知所访问的页面，对其进行渲染并将结果返回给浏览器；而当使用 HashLocationStrategy 策略时，由于 hash 部分不会发送到服务器，所以各页面请求的都是同一个 URL，导致服务器无法通过 URL 得知所要访问的页面，也就无从进行渲染了。

要使用 PathLocationStrategy 路由策略，必须满足三个条件：

第一，浏览器需要支持 HTML 5 的 `history.pushState()` 方法，正是这一方法使得 RouterLink 指令在跳转时即使更改了 URL 的 path 部分，也依然不会引起页面刷新。

第二，需要在服务器上进行设置，将应用的所有 URL 重定向到应用的首页。这是因为该策略所生成的 URL（如 `http://localhost:3000/list`）在服务器上并不存在与其相对应的文件结构，如果不进行重定向，服务器将返回 404 错误。

第三，需要为应用设置一个 base 路径，Angular 将以 base 路径为前缀来生成和解析 URL。这样做的好处是服务器可以根据 base 路径来区分来自不同应用的请求。

如何在服务器上进行重定向设置超出了 Angular 的范畴，这里就不深入讲解了，接下来只对设置 base 路径的两种方式加以介绍。

第一种方式，如 11.2 节中所述，是通过设置 index.html 页面中 <base> 标签的 href 属性来完成的。我们把通讯录例子的 <base> 标签修改一下，将应用的 base 路径变为 app，那么相应的联系人列表页的 URL 也将变为 http://localhost:3000/app/list。

```
<!-- index.html -->
<head>
  <base href="/app">
  <!-- ... -->
</head>
```

第二种方式，是通过向应用中注入 APP_BASE_HREF 变量来实现的，同样将应用的 base 路径设成了 app。示例代码如下：

```
// app.module.ts
import { Component, NgModule } from '@angular/core';
import { APP_BASE_HREF } from '@angular/common';

@NgModule({
  providers: [{provide: APP_BASE_HREF, useValue: '/app'}],
  //...
})
export class AppModule {}
```

但使如果这两种方式同时使用，第二种则具有更高的优先级。

11.4 路由跳转

Web 应用中的页面跳转，指的是应用响应某个事件，从一个页面跳转到另一个页面的行为。对于使用 Angular 构建的单页应用而言，页面跳转实质上就是从配置项跳转到另一个配置项的行为。页面跳转流程如图 11-6 所示，当某个事件引发了跳转时，Angular 会根据跳转时的参数生成一个 UrlTree 实例来和配置项进行匹配，如果匹配成功，则显示相应的组件并将新 URL 更新在浏览器地址栏中；如果匹配不成功，则报错。

本节将对 Angular 应用中进行页面跳转的两种方式进行介绍。

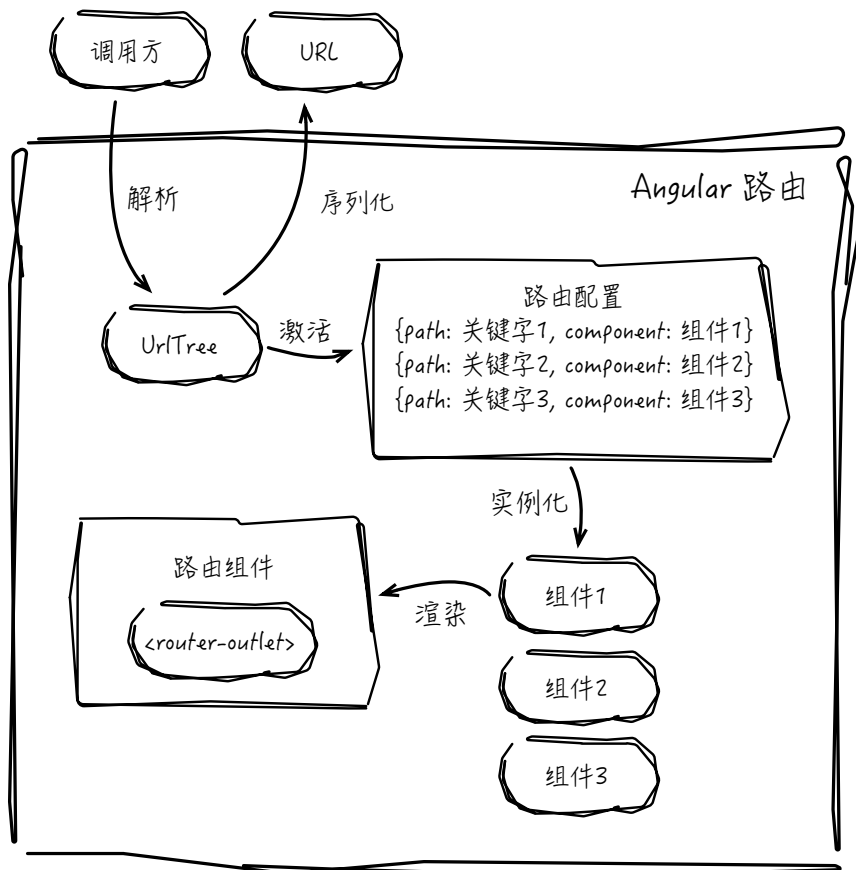


图 11-6 页面跳转流程

11.4.1 使用指令跳转

指令跳转通过使用 RouterLink 指令来完成。该指令接收一个链接参数数组，Angular 将根据该数组来生成 UrlTree 实例进行跳转。通讯录例子中的 FooterComponent 组件将 RouterLink 指令应用在 <a> 标签上，分别为联系人列表页和收藏页定义了两个超链接，当用户单击超链接时便会跳转到相应的页面。

```
<!-- app/shared/footer.component.html -->
<nav>
  <!-- http://localhost:3000/collection -->
  <a [routerLink]="['/collection']">
    <i>收藏</i>
  </a>
```

```

<!-- http://localhost:3000/list -->
<a [routerLink]="['/list']">
  <i>通讯录</i>
</a>
</nav>

```

如果不借助于 RouterLink 指令而以纯 HTML 的方式来定义超链接，所导致的结果是单击超链接后会使得整个页面被重新加载。示例代码如下：

```

<nav>
  <a href="/collection">
    <i>收藏</i>
  </a>
  <a href="/list">
    <i>通讯录</i>
  </a>
</nav>

```

通过分析下面 RouterLink 指令的部分源码可知，Angular 通过以下两个步骤来保证在不重新加载应用的情况下完成跳转。

- 在 click 事件中调用 preventDefault() 方法来禁止单击 <a> 标签后向服务器发送请求的行为，从而避免了跳转加载。
- 调用 Router.navigateByUrl() 方法来启动跳转流程。

```

@HostListener('click', ['$event.button', '$event.ctrlKey', '$event.metaKey'])
onClick(button: number, ctrlKey: boolean, metaKey: boolean): boolean {
  //...

  this.router.navigateByUrl(this.urlTree); // 跳转到指定页面，渲染相应组件
  return false; // 当 HostListener 装饰的回调函数返回 false 时，
                // Angular 会调用 `preventDefault()` 方法
}

```

RouterLink 指令的另一个强大之处在于它可以被应用在任何 HTML 元素上，使得页面跳转不需要依赖超链接。下面的模板使用了 <button> 标签来代替 <a> 标签，单击各按钮后页面跳转依然正常工作。示例代码如下：

```

<nav>
  <button [routerLink]="['/collection']">
    <i>收藏</i>
  </button>

```



```
<button [routerLink]="['/list']">
  <i>通讯录</i>
</button>
</nav>
```

此外,当 RouterLink 被激活时,还可以通过 RouterLinkActive 指令为其相应的 HTML 元素指定 CSS 类。下面的例子定义了一个 CSS 类 .active, 并通过 routerLinkActive 将其赋给收藏页的链接。当单击该链接后, .active 类将被应用到 <a> 标签上。示例代码如下:

```
/* footer.component.css */
.active {
  background-color: #3DD689;
}

<!-- footer.component.html -->
<nav>
  <a [routerLink]="['/collection']" routerLinkActive="active">
    <i>收藏</i>
  </a>
  <!-- ... -->
</nav>
```

RouterLinkActive 指令除可以作用于 routerLink 所在的元素之外,还可以作用于这些元素的任意祖先元素。当该祖先元素下的任意 routerLink 处于激活状态时,该祖先元素都将获得 routerLinkActive 指定的 CSS 类。下面的例子不管当前是处于联系人列表页还是收藏页, <nav> 标签都将获得 .active 类。示例代码如下:

```
<nav routerLinkActive="active">
  <a [routerLink]="['/collection']">
    <i>收藏</i>
  </a>
  <a [routerLink]="['/list']">
    <i>通讯录</i>
  </a>
</nav>
```

11.4.2 使用代码跳转

从上文 RouterLink 指令的部分源码可以看出,跳转流程是通过调用 Router.navigateByUrl() 方法来启动的。RouterLink 指令仅响应 click 事件,如果需要响应其他事件或者需要根据运行时的数据动态决定如何跳转,则可以通过调用 Router.navigateByUrl() 或

其兄弟方法 `Router.navigate()` 来完成。下面的例子实现了在进入联系人列表页 1 秒后自动跳转到收藏页的功能。示例代码如下：

```
// list.component.ts
import { Router } from '@angular/router';

//...
export class ListComponent implements OnInit {
  constructor(private _router: Router) {
    setTimeout(()=>{
      _router.navigateByUrl('/collection');
      // 或者执行: _router.navigate(['/collection']);
    }, 1000);
  }
}
```

通过分析源码可知, `Router.navigateByUrl()` 和 `Router.navigate()` 方法的底层工作机制基本一致, 最终都是通过调用 `Router.scheduleNavigation()` 方法来执行跳转的。不同的地方在于两个方法指定跳转的目标配置项的方式不同, 其中 `Router.navigateByUrl()` 方法通过一个 URL 字符串或 `UrlTree` 实例来指定。示例代码如下:

```
navigateByUrl(url: string|UrlTree, extras: NavigationExtras = {skipLocationChange:
  false}): Promise<boolean> {
  if (url instanceof UrlTree) {
    return this.scheduleNavigation(url, extras);
  } else {
    // 解析 URL 字符串生成相应的 UrlTree 实例
    const urlTree = this.urlSerializer.parse(url);
    // 使用 UrlTree 实例进行跳转
    return this.scheduleNavigation(urlTree, extras);
  }
}
```

而 `Router.navigate()` 方法与 `RouterLink` 指令相似, 通过链接参数数组来指定。示例代码如下:

```
navigate(commands: any[], extras: NavigationExtras = {skipLocationChange: false}):
  Promise<boolean> {
  // 通过链接参数数组生成 UrlTree 实例
  return this.scheduleNavigation(this.createUrlTree(commands, extras), extras);
}
```

这两个方法除可以通过第一个参数来指定目标配置项外，还支持用 `extras` 参数定义跳转的具体行为。例如，如果想在不变 URL 的情况下完成跳转，则可以通过以下代码来完成：

```
_router.navigateByUrl('/collection', {skipLocationChange: true});
```

关于 `extras` 参数的其他用法，感兴趣的读者可以参考官方文档来了解更多的内容，在此不再赘述。

11.5 路由参数

在“组件”章节中介绍了如何使用 `@Input` 装饰器向组件传递数据，除此之外，Angular 路由还提供了路由参数的功能，允许通过 URL 向组件传递数据。在通讯录例子中，如果想通过 URL 对不同的联系人进行区分，一种简单的实现方式是将联系人的 `id` 添加到 URL 中，此后联系人详情页通过提取 URL 中的联系人 `id` 便可以进一步获取该联系人的详情并予以显示。本节将对 Angular 路由获取 URL 参数的几种方式分别进行介绍，包括 Path 参数和 Query 参数。

11.5.1 Path 参数

顾名思义，Path 参数是通过解析 URL 的 `path` 部分来获取参数的。在定义一个配置项的 `path` 属性时，可以使用“/”字符来对 `path` 属性进行分段，如果一个分段以“:”字符开头，则 URL 中与该分段进行匹配的部分将作为参数传递到组件中。下面的代码为联系人详情页的路由配置项，其定义了一个名为 `id` 的 Path 参数，对于 `http://localhost:3000/detail/1`，参数 `id` 的值为 1；对于 `http://localhost:3000/detail/2`，参数 `id` 的值则为 2；依此类推。

```
// app.routes.ts
export const ContactsAppRoutes: RouterConfig = [
  { path: 'detail/:id', component: DetailComponent }
];
```

值得注意的是，只有当 URL 解析出来的分段数和 `path` 属性的分段数一致时，才能匹配到该配置项。本例中 `path` 属性的分段数为 2，因此下面两个 URL 无法匹配到该配置项。

```
http://localhost:3000/detail    # 分段数为1
http://localhost:3000/detail/1/segment  # 分段数为3
```

给路由参数赋值，除可以直接在浏览器地址栏中输入 URL 外，还可以通过 Router-Link 指令或者跳转方法来完成：

```
// Angular 会将链接参数数组的每一个非对象元素当成一个分段进行拼接，  
// 因此下面的链接参数数组对应的 path 为 `detail/1`
```

```
<a [routerLink]=["/detail", 1]>  
_router.navigate(['/detail', 1]);
```

```
// 或者直接指定 path  
_router.navigateByUrl('detail/1');
```

在组件中获取 Path 参数，需要导入 ActivatedRoute 服务，该服务提供了两种方式，分别适用于不同页面间跳转和同一页面内跳转。

Angular 应用从一个页面跳转到另一个新的页面，实质上是从一个配置项跳转到另一个配置项。在这个过程中，Angular 除会为配置项所对应的组件创建实例外，还会为该配置项本身创建一个 ActivatedRoute 实例来表示该配置项已被激活。该 ActivatedRoute 实例包含了一个快照（即 snapshot 属性），记录了从当前 URL 中解析出来的所有 Path 参数。下面展示了通讯录例子中的 DetailComponent 组件是如何通过快照来获取 Path 参数的。示例代码如下：

```
// detail.component.ts  
// 1. 导入 ActivatedRoute 服务  
import { ActivatedRoute } from '@angular/router';  
// ...  
export class DetailComponent implements OnInit, OnDestroy {  
  contact_id: string;  
  
  constructor( private _activatedRoute:ActivatedRoute ) {  
    console.log('创建 DetailComponent 组件实例');  
  }  
  
  ngOnInit() {  
    // 2. 通过快照获取 Path 参数  
    this.contact_id = this._activatedRoute.snapshot.params['id'];  
    console.log('参数id的值为: '+this.contact_id);  
  }  
  
  //...  
}
```

此时通过 `http://localhost:3000/detail/1` 直接访问联系人详情页，可以在浏览器控制台上看到如下输出，则表示通过快照获取到的值是正确的。

创建 `DetailComponent` 组件实例

参数 `id` 的值为：1

接下来介绍如何在通讯录例子页面跳转时获取参数值。首先在联系人详情组件的模板 `detail.component.html` 上添加一个链接，希望达到的效果是当单击该链接后，能够显示下一名联系人的信息。示例代码如下：

```
<!-- detail.component.html -->
<div class="detail-contain">
  <div class="header-detail">
    <a [routerLink]="['']" class="back-to-list">
      <i class="icon-back"></i>
      所有联系人
    </a>
    <a [routerLink]="['/detail', nextContactId()]" class="back-to-list">
      下一联系人
    </a>
  <!-- ... -->
</div>
<!-- ... -->
</div>
```

`nextContactId()` 方法通过简单地加 1 来获取下一名联系人的 `id`：

```
// detail.component.ts
// ...
export class DetailComponent implements OnInit, OnDestroy {
  //...
  nextContactId() {
    return parseInt(this.contact_id) + 1;
  }
}
```

假设当前 URL 为 `http://localhost:3000/detail/1`，在单击“下一联系人”链接后，URL 按照预期变成了 `http://localhost:3000/detail/2`，但是页面上显示的仍然是原先联系人的信息。这是因为 Angular 在处理同一页面内跳转时，不会重新创建组件的实例，所以组件的构造函数和 `ngOnInit()` 方法都没有被调用到。虽然 Angular 会将快照中参数 `id` 的值更新为 2，但没有将这个更新通知到组件。为了解决这个问题，`ActivatedRoute` 服务提供了一个 `Observable` 对象，允许对参数的更新进行订阅。示例代码如下：

```
// detail.component.ts
// ...
export class DetailComponent implements OnInit, OnDestroy {
  contact_id: string;
  private sub: any;
  //...

  ngOnInit() {
    this.sub = this._activatedRoute.params.subscribe(params => {
      this.contact_id = params['id'];
      console.log('参数id的值为: '+this.contact_id);

      this.getById(this.contact_id);
    });
  }

  ngOnDestroy() {
    // 为了避免内存泄漏，在组件销毁时应该取消订阅
    this.sub.unsubscribe();
  }

  //...
}
```

此时单击链接，便可以显示出下一位联系人的信息。

11.5.2 Query 参数

我们也可以通过解析 URL 的 query 部分来获取参数值。由于 URL 的 query 部分不用于和配置项进行匹配，因此每一个配置项都可以拥有任意多个查询参数。下面的 URL 给联系人列表页定义了一个查询参数，表示只希望在页面上显示 5 位联系人。

`http://localhost:3000/list?limit=5`

与 Path 参数类似，Query 参数同样可以通过 RouterLink 指令或者跳转方法来赋值。示例代码如下：

```
<a [routerLink]="['/list']" [queryParams]="{limit: 5}">

this._router.navigate(['/list'], {queryParams: {limit: 5}});
this._router.navigateByUrl('/list?limit=5');
```

Query 参数的获取，需要借助于 `ActivatedRoute` 服务提供的 `Observable` 类型对象 `queryParams` 来完成。下面的代码通过获取 Query 参数来对显示在页面上的联系人数目进行限制。

```
// list.component.ts
import { ActivatedRoute } from '@angular/router';
//...

export class ListComponent implements OnInit, OnDestroy {
  contacts: any[];
  private limit: number;
  private sub: any;

  constructor(private _activatedRoute: ActivatedRoute) { }

  ngOnInit() {
    this.getContacts();
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }

  getContacts() {
    //...

    this.sub = this._activatedRoute.queryParams.subscribe(params => {
      this.limit = parseInt(params['limit']);

      if (this.limit) {
        this.contacts.splice(this.limit);
      }
    });
  }

  //...
}
```

11.5.3 Matrix 参数

页面上所有组件都可以访问 Query 参数的内容，如果想精准地向某一个组件传递参数，则需要使用 Matrix 参数。详细内容将在 11.6 节的“子路由”部分展开。

11.6 子路由和附属 Outlet

11.6.1 子路由

基本用法

在“组件”章节中介绍到，一个组件可以被嵌入到另外一个组件中，从而建立起组件之间的多级嵌套关系。与此类似，Angular 也允许一个路由组件被嵌入到另一个路由组件中，从而建立路由的多级嵌套关系。如图 11-7 所示，假设通讯录例子中的 DetailComponent 组件本身是一个路由组件，在其中可以显示关于个人简介的 AnnotationComponent 组件和相册 AlbumComponent 组件，同时其也作为一个子路由组件被嵌入到根路由组件 ContactApp 之中。

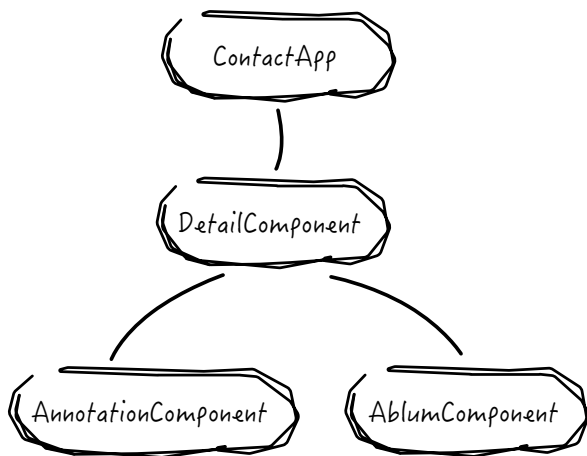


图 11-7 子路由

这种嵌套关系通过如下路由配置来建立，其中 children 是专门服务于子路由组件 DetailComponent 的路由配置，其所包含的每一个配置项的 path 值都相对于该子路由组件的 path 值。


```
// app.routes.ts
export const rootRouterConfig: Routes = [
  { path: 'detail/:id', component: DetailComponent,
    children: [
      { path: '', component: Annotation }, // http://localhost:3000/detail/:id
      { path: 'album', component: Album } // http://localhost:3000/detail/:id/album
    ]
  }
];
```

在浏览器地址栏中输入 `http://localhost:3000/detail/1/album`, 可以看到 `DetailComponent` 组件和 `AlbumComponent` 组件的内容都在联系人详情页上显示出来。示例代码如下:

```
<!-- detail.component.html -->
<detail>
  <!-- ... -->
  <router-outlet></router-outlet>
  <album>
    <!-- ... -->
  </album>
</detail>
```

Matrix 参数

现在假设需要实现一个功能, 要求 `AlbumComponent` 组件能够根据搜索条件来显示照片, 这就需要将搜索条件作为参数传递给 `AlbumComponent` 组件。Path 参数无法满足这个功能, 因为 Path 参数是确定的且必须为每一个参数赋值才能匹配到配置项, 而搜索条件是不确定的 (比如既可以指定返回最新的 5 张照片, 也可以通过指定开始时间和结束时间来返回 2016 年的所有照片)。虽然 Query 参数可以用来传递不确定的参数, 但缺点在于 Query 参数是一块公共的区域, 页面上的所有组件都可以访问。比如对于以下 URL, `DetailComponent` 组件和 `AlbumComponent` 组件都可以获取到相同的 `before` 和 `after` 参数值:

`http://localhost:3000/detail/6/album?after=2016-01-01&before=2016-12-31`

一方面, 在本例中 `DetailComponent` 组件并不关心 `AlbumComponent` 组件的搜索条件; 另一方面, 假如 `DetailComponent` 组件也希望接收 `before` 和 `after` 两个参数, 但是参数值与传递给 `AlbumComponent` 组件的不一样, 此时使用 Query 参数亦无法满足需求,

所以更理想的方式是能够将参数精准地传递给所需要的组件。为了实现参数的精准传递, Angular 提供了 Matrix 参数, 它通过在链接参数数组中插入一个对象来进行赋值。示例代码如下:

```
<a [routerLink]="['/detail', this.contact_id, {after:'2015-01-01', before:'2015-12-31'}, 'album', {after:'2016-01-01', before:'2016-12-31'}]">Link</a>
```

Angular 会将该对象的属性转化为以 “;” 为分隔符的键值对, 拼接到与该对象左边最近的 URL 分段上。依据上述链接参数数组生成的 URL 如下, DetailComponent 组件和 AlbumComponent 组件都将获得不同的参数值:

```
http://localhost:3000/detail/6;after=2015-01-01;before=2015-12-31/album;after=2016-01-01;before=2016-12-31
```

这种在一个 URL 分段内使用 “;” 分隔键值对的方式称为 Matrix URI, 由互联网之父 Tim Berners-Lee 于 1996 年提出。根据其定义, 每一个 URL 分段都可以拥有任意多个键值对, 每个键值对只为其所在的分段服务。虽然 Matrix URI 一直没有进入 HTML 标准, 但它能够清晰地表示出每一个 URL 分段所具有的键值对。Angular 利用这个特性, 将 Matrix 参数精准地传递给分段所对应的组件。Matrix 参数的获取方式和 Path 参数一样, 可以通过 ActivatedRoute 服务提供的快照和 Observable 对象两种方式来获取, 在此不再赘述。

11.6.2 附属 Outlet

Angular 允许一个路由组件包含多个 Outlet, 从而可以在一个路由组件中同时显示多个组件。其中, 主要 Outlet (Primary Outlet) 有且仅有一个, 附属 Outlet (Auxiliary Outlet) 可以有任意多个, 各个附属 Outlet 通过不同的命名加以区分。每一个 Outlet 均可以通过路由配置来指定其可以显示的组件, 这使得 Angular 可以灵活地对各个组件进行组合, 从而满足不同场景的需求。在通讯录例子中, 假设有灵活地在 DetailComponent 组件中控制 AnnotationComponent 组件和相册 AlbumComponent 组件的显示, 那么首先可以在 DetailComponent 组件的模板中添加一个名为 aux 的附属 Outlet。示例代码如下:

```
<!--detail.component.html -->
<div class="detail-contain">
  <!-- ... -->
  <router-outlet></router-outlet>
  <router-outlet name="aux"></router-outlet>
</div>
```

接下来在路由配置中定义在主要 Outlet 和附属 Outlet 上均可以显示 Annotation-Component 组件和相册 AlbumComponent 组件。

```
// app.routes.ts
export const rootRouterConfig: Routes = [
  { path: 'detail/:id', component: DetailComponent,
    children: [
      // 主要 Outlet
      { path: '', component: AnnotationComponent },
      { path: 'album', component: AlbumComponent },
      // 附属 Outlet
      { path: 'annotation', component: AnnotationComponent, outlet: 'aux'},
      { path: 'album', component: AlbumComponent, outlet: 'aux'},
    ]
  }
];
```

以 id 为 1 的联系人为例，表 11-1 列出了各种可能的组合及其对应的 URL 和链接参数数组。在链接参数数组中，如果一个元素包含了 outlets 属性，则表示该元素将用于为各个 Outlet 进行配置项匹配。

表 11-1 主要 Outlet 和附属 Outlet 各组合的定义及其对应的 URL 和链接参数数组

组件组合	URL Path	链接参数数组
主要 Outlet: AnnotationComponent 附属 Outlet: 无	/detail/1	['/detail', 1]
主要 Outlet: AnnotationComponent 附属 Outlet: AnnotationComponent	/detail/1/(aux:annotation)	['/detail', 1, {outlets: {aux: ['annotation']}}]
主要 Outlet: AnnotationComponent 附属 Outlet: AlbumComponent	/detail/1/(aux:album)	['/detail', 1, {outlets: {aux: ['album']}}]
主要 Outlet: AlbumComponent 附属 Outlet: 无	/detail/1/album	['/detail', 1, 'album']
主要 Outlet: AlbumComponent 附属 Outlet: AnnotationComponent	/detail/1/(album//aux:annotation)	['/detail', 1, {outlets: {primary: ['album'], aux: ['annotation']}}]
主要 Outlet: AlbumComponent 附属 Outlet: AlbumComponent	/detail/1/(album//aux:album)	['/detail', 1, {outlets: {primary: ['album'], aux: ['album']}}]

11.7 路由拦截

Angular 的路由拦截允许在从一个配置项跳转到另一个配置项之前执行指定的逻辑，并根据执行的结果来决定是否进行跳转。Angular 提供了五类路由拦截：

- CanActivate，激活拦截。
- CanActivateChild，与 CanActivate 类似，用于控制是否允许激活子路由配置项。
- CanDeactivate，反激活拦截。
- Resolve，数据预加载拦截。
- CanLoad，模块加载拦截。

本节将以通讯录例子的编辑页为例，展示 CanActivate、CanActivateChild、CanDeactivate 及 Resolve 这几类路由拦截的用法。

11.7.1 激活拦截与反激活拦截

激活拦截与反激活拦截用于控制是否可以激活或反激活目标配置项，其工作流程如图 11-8 所示。

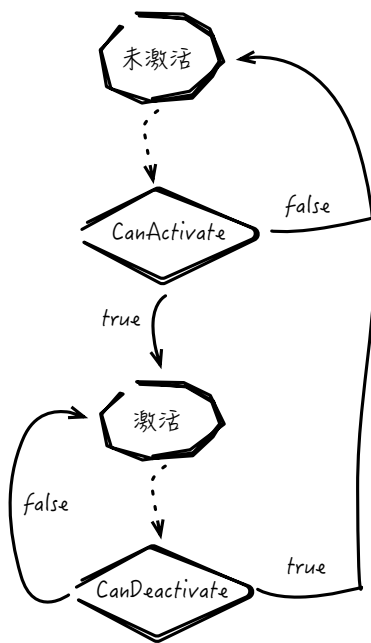


图 11-8 激活拦截与反激活拦截工作流程

CanActivate

在通讯录例子中，假设需要根据用户的登录状态来决定能否访问联系人编辑页。要实现这个功能，可以通过为联系人编辑页添加一个判断登录状态的 CanActivate 拦截来实现。

首先，通过实现 CanActivate 接口创建拦截服务。该接口只包含了一个 canActivate() 方法，最简单的情况是，当该方法返回 true 时，表示允许通过 CanActivate 拦截；当返回 false 时，则表示不允许通过 CanActivate 拦截，对目标配置项不予激活。

```
// can-activate-guard.ts
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';

@Injectable()
export class CanActivateGuard implements CanActivate {
  canActivate() {
    if(/*用户已登录*/){
      return true;
    }
    else{
      return false;
    }
  }
}
```

然后，在目标配置项中指定上面创建的服务作为其 CanActivate 拦截服务。

```
//app.routes.ts
import { CanActivateGuard } from "../services/can-activate-guard";

export const rootRouterConfig: Routes = [{
  path: 'operate/id/:id',
  component: OperateComponent,
  canActivate: [CanActivateGuard]
}];
```

最后，将该服务注入到应用中。

```
//app.module.ts
import { CanActivateGuard } from "../services/can-activate-guard";
```

```
@NgModule({
  //...
  providers:[CanActivateGuard]
})
export class AppModule {}
```

除了返回布尔值，`canActivate()` 方法还可以返回一个 `Observable` 对象，当该对象触发（emit）`true` 时，表示允许通过拦截；当触发 `false` 时则表示不允许通过拦截。这个特性使得 `CanActivate` 拦截可以根据异步处理的结果来进行判断。

```
//can-activate-guard.ts
// ...
@Injectable()
export class CanActivateGuard implements CanActivate {
  canActivate() {
    return new Observable<boolean>(observer => {
      observer.next(true);
      observer.complete();
    });
  }
}
```

此外，Angular 还会给 `canActivate()` 方法传递两个参数。

- `ActivatedRouteSnapshot`，表示所要激活的目标配置项，可以通过它访问配置项的相关信息。
- `RouterStateSnapshot`，表示应用当前所处的路由状态，其包含了当前所需的所有配置项。

关于这两个参数的用法示例如下：

```
//can-activate-guard.ts
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/
router';

@Injectable()
export class CanActivateGuard implements CanActivate {
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

    // 获取配置项信息
    console.log(route.routeConfig);
```

```

// RouterStateSnapshot 按照路由配置中的定义，将所需的配置项以树形结构方式
  组织起来
console.log(state.root);

return true;
}
}

```

CanActivateChild

CanActivateChild 拦截用于控制是否允许激活子路由配置项，其用法与 CanActivate 拦截相似，在此不再赘述。

CanDeactivate

在通讯录例子的编辑页中，当用户单击“取消”按钮时，可以通过在 CanDeactivate 拦截中判断联系人信息是否被修改且未保存来决定是否允许离开编辑页。使用 CanDeactivate 拦截的用法可分为以下三步：

首先，通过实现 CanDeactivate 接口创建拦截服务。该接口只包含了一个 canDeactivate() 方法，该方法除第一个参数为目标配置项对应组件的实例外，其余参数的使用方式与 canActivate() 方法一样。下面的例子通过调用组件实例的 isModified() 方法来判断组件内容是否发生修改且未保存。

```

// can-deactivate-guard.ts
import { Injectable } from '@angular/core';
import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '
  @angular/router';

@Injectable()
export class CanDeactivateGuard implements CanDeactivate<any> {
  canDeactivate(component: any, route: ActivatedRouteSnapshot, state:
    RouterStateSnapshot) {
    if(component.isModified()){
      return true;
    }
    else{
      return false;
    }
  }
}
}

```

然后，在目标配置项中指定该服务作为其 CanDeactivate 拦截服务。

```
//app.routes.ts
import { CanDeactivateGuard } from "../services/can-deactivate-guard";

export const rootRouterConfig: Routes = [{
  path: "operate/id/:id",
  component: Operate,
  canActivate: [CanActivateGuard],
  canDeactivate: [CanDeactivateGuard]
}];
```

最后，将该服务注入到应用中。

```
// app.module.ts
import { CanDeactivateGuard } from "../services/can-deactivate-guard";

@NgModule({
  //...
  providers: [CanDeactivateGuard]
})
export class AppModule {}
```

11.7.2 数据预加载拦截

数据预加载拦截（Resolve 拦截）适用于对数据进行预加载，当确认数据加载成功后，再激活目标配置项。其工作流程如图 11-9 所示。

下面将展示如何在通讯录例子的联系人编辑页显示前对联系人信息进行预加载。该过程可分为四步：

首先，通过实现 `Resolve<T>` 泛型接口创建拦截服务。该服务只有一个 `resolve()` 方法，用于执行数据预加载逻辑。该方法可以直接将数据返回，在异步情况下也可以通过 `Observable` 对象触发。值得注意的是，所返回的任何数据（包括 `false`）都将存放于配置项的 `data` 参数部分，如果没有预加载到所期望的数据，则只能通过代码跳转的方式来达到不激活目标配置项的目的。示例代码如下：

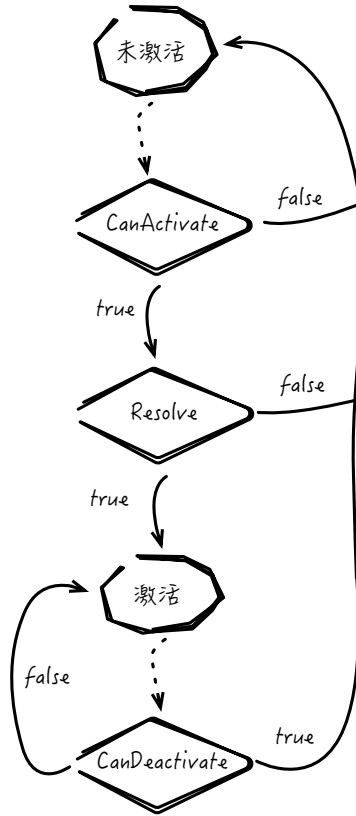


图 11-9 数据预加载拦截工作流程

```

// resolve-guard.ts
import { Injectable } from '@angular/core';
import { Router, Resolve, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { ContactService } from '../services/contact.service';

@Injectable()
export class ResolveGuard implements Resolve<any> {
  contacts: {};
  constructor(private _router: Router, private _contactService: ContactService){}

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    // 返回Observable对象
    return this._contactService.getContactById(route.params['id']).map(
      res => {

```

```
        if(res){
            return res;
        }
        else{
            // 预加载失败，代码跳转至其他配置项
            this._router.navigate(['/list']);
        }
    }
});
}
```

然后，在目标配置项中指定该服务作为其 Resolve 拦截服务。下面的配置表示通过 ResolveGuard 服务预加载的数据将存放于 data 参数的 contact 属性下。

```
// app.routes.ts
import { ResolveGuard } from "../services/resolve-guard";

export const rootRouterConfig: Routes = [{
    path: "operate/id/:id",
    component: Operate,
    canActivate: [CanActivateGuard],
    resolve: {
        contact: ResolveGuard
    }
}];
```

接下来，将该服务注入到应用中。

```
// app.module.ts
import { ResolveGuard } from "../services/resolve-guard";

@NgModule({
    // ...
    providers : [ResolveGuard]
})
export class AppModule {}
```

最后，在目标配置项所指定的组件中访问预加载的数据。

```
// operate.component.ts
// ...
export class OperateComponent implements OnInit {
  constructor(private _activatedRoute:ActivatedRoute) {}

  ngOnInit() {
    this._activatedRoute.data.subscribe(data=>{
      console.log(data.contact);
    });
  }
}
```

11.8 模块的延迟加载

前文提到，Angular 应用由一个根模块和任意多个特性模块组成。一个大型 Web 应用通常会包含为数不少的特性模块，如果在首屏加载时便将所有的特性模块加载进来，对于用户体验和服务器负载均会有所影响。为此，Angular 路由提供了对特性模块进行延迟加载的支持，使得只有在真正需要某一个模块的时候，才将其加载进来。

11.8.1 延迟加载实现

通讯录例子只有一个根模块，为了演示如何进行特性模块的延迟加载，下面将 OperateComponent 组件从根模块中抽取出来，单独为其创建一个 OperateModule 模块。与根模块需要初始化各项路由服务不同，特性模块仅需要对其路由配置进行解析，因此子路由模块通过调用 RouterModule.forChild() 方法来创建。示例代码如下：

```
// operate.module.ts
import { NgModule } from '@angular/core';
import { FormsModule } from "@angular/forms";
import { BrowserModule } from "@angular/platform-browser";
import { Routes, RouterModule } from '@angular/router';
import { OperateComponent } from '../widget/operate.component';
import { ContactService } from "../services/contact.service";

const operateRoutes: Routes = [
  { path: "id/:id", component: OperateComponent },
  { path: "isAdd/:isAdd", component: OperateComponent }
];

@NgModule({
```

```
imports : [BrowserModule, FormsModule, RouterModule.forChild(operateRoutes)],  
declarations: [OperateComponent],  
providers: [ContactService]  
})  
export class OperateModule {}
```

此后 OperateComponent 组件便不再需要在根模块 AppModule 中导入。示例代码如下：

```
// app.module.ts  
@NgModule({  
  declarations: [  
    // OperateComponent,  
    // ...  
  ],  
  // ...  
})  
export class AppModule {}
```

最后，还需要对根模块的路由配置进行修改。示例代码如下：

```
// app.routes.ts  
// ...  
export const rootRouterConfig: Routes = [  
  // OperateComponent 组件的配置项已在 OperateModule 模块中定义，故在此删除  
  // { path: "operate/:id", component: OperateComponent },  
  // { path: "operate/isAdd/:isAdd", component: OperateComponent },  
  
  { path: 'operate', loadChildren: 'app/router/operate.module.ts#OperateModule' }  
];
```

loadChildren 指定了延迟加载模块的路径，井号“#”后面的表示模块类名。当用户访问地址 /operate 时，Angular 才会加载 operate.module.ts 这个模块。



SystemJS 是模块加载器，可以导入任何流行格式的模块（如 CommonJS、UMD、AMD、ES 6 等），工作在 ES 6 模块加载 polyfill 之上，能够很好地处理和检测所使用的格式。Angular CLI 提供了一些专用工具，在编译阶段会分析 loadChildren 指定模块，并对该模块单独打包生成独立的代码文件，然后通过内置的加载器加载这个单独的模块文件。

11.8.2 模块预加载

延迟加载使得首屏加载的资源包的大小减小很多，这些模块只在用户触发的时候才开始加载。但对于某些模块来说，触发时才加载可能不是最优的解决方案。这样的模块虽然不需要首屏加载，但可能有很大的概率用户会访问使用到，因此最好不用等待用户触发，而是在首屏资源加载完后立即加载，这种加载模式就叫作预加载。

预加载的模块首先得是一个延迟加载的模块，沿用延迟加载的例子，对 `RouterModule` 实现预加载。让所有延迟加载的模块加上预加载功能非常简单，只需在根模块的 `RouterModule` 中添加一个 `preloadingStrategy` 配置项即可。示例代码如下：

```
RouterModule.forRoot(  
  rootRouterConfig,  
  { preloadingStrategy: PreloadAllModules }  
)
```

加上这个配置后，所有的延迟加载模块将不再等待用户触发，而是等待首屏资源加载完后立即加载。

不过，这样的配置显然不够灵活，更好的方式是对预加载的策略做自定义配置。开发者可以通过实现 `Angular` 提供的 `PreloadingStrategy` 接口自定义预加载策略。

首先定义一个服务，并实现 `PreloadingStrategy` 接口。示例代码如下：

```
import { Injectable } from '@angular/core';  
import { PreloadingStrategy, Route } from '@angular/router';  
  
@Injectable()  
export class MyPreloadingStrategy implements PreloadingStrategy {  
  preload(route: Route, load: () => Observable<any>): Observable<any> {  
    return Observable.of(null);  
  }  
}
```

`preload()` 方法的返回类型必须是一个 `Observable` 对象，`Angular` 会遍历每一个 `route` 对象并执行 `preload()` 函数，以此来判断该 `route` 对应的模块是否需要预加载。它接受两个参数：

- `route`：当前处理中的 `route` 对象。
- `load`：内置异步模块加载器函数。

上面这个例子直接返回 `Observable.of(null)`, 表示不进行预加载。`MyPreloadingStrategy` 这个服务的目的是进行有选择的预加载, 可以根据 `route` 对象里的 `data` 属性提供的信息进行判断。示例代码如下:

```
// ...
export class MyPreloadingStrategy implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    if (route.data && route.data['preload']) {
      return load();
    } else {
      return Observable.of(null);
    }
  }
}
```

如果 `data` 对象里设置了 `preload` 为 `true`, `preload` 函数即返回 `load()` 加载器函数, 这表示该路由对应的模块需要进行预加载。

这个 `MyPreloadingStrategy` 服务已经完成了, 下面需要把原来的 `PreloadAllModules` 替换成新的 `MyPreloadingStrategy`。示例代码如下:

```
RouterModule.forRoot(
  rootRouterConfig,
  { preloadingStrategy: MyPreloadingStrategy }
)
```

然后依据这个规则, 控制模块预加载就变得非常简单了。以 `OperateModule` 为例, 调整后的示例代码如下:

```
export const rootRouterConfig: Routes = [
  {
    path: 'operate',
    loadChildren: 'app/router/operate.module.ts#OperateModule',
    data: { preload: true } // 自定义预加载策略
  }
];
```

在这个 `route` 对象里设置 `preload` 为 `true`后, `OperateModule` 的加载方式由原来的延迟加载变更为预加载, 而其他延迟加载模块并不会受到影响, 还是会等待用户触发时才加载。

11.8.3 模块加载拦截

在默认情况下，如果 URL 匹配到延迟加载的配置项，相应的特性模块便会被加载进来。如果想动态判断是否对该模块进行加载，则可以使用 CanLoad 拦截，其工作流程如图 11-10 所示。

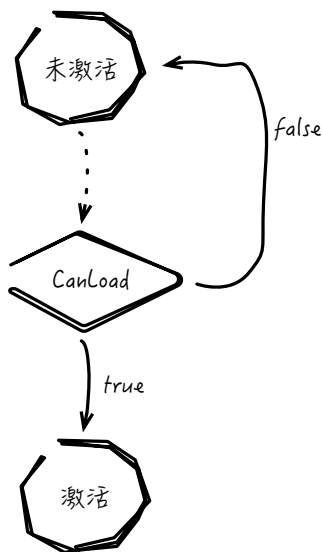


图 11-10 CanLoad 拦截工作流程

CanLoad 拦截的用法和 CanActivate 等其他拦截类似，首先需要实现 CanLoad 接口来创建拦截服务。由于在触发 CanLoad 拦截时，相应的特性模块还未被加载，因此能传递给 canLoad() 方法的只有延迟加载配置项的信息。示例代码如下：

```
// can-load-guard.ts
import { Injectable } from '@angular/core';
import { CanLoad, Route } from '@angular/router';

@Injectable()
export class CanLoadGuard implements CanLoad {
  canLoad(route: Route) {

    // route 参数为延迟加载配置项
    console.log(route.path); // 输出: operate

    if(/* 允许加载 */){
```

```
        return true;
    }
    else{
        return false;
    }
}
}
```

接着，在延迟加载配置项中指定 CanLoad 拦截服务。示例代码如下：

```
// app.routes.ts
import { CanLoadGuard } from '../services/can-load-guard';

export const rootRouterConfig: Routes = [{
  path: 'operate',
  loadChildren: 'app/router/operate.module.ts#OperateModule',
  canLoad: [CanLoadGuard]
}];
```

最后，将 CanLoad 拦截服务注入到根模块中。示例代码如下：

```
// app.module.ts
import { CanLoadGuard } from "../services/can-load-guard";

@NgModule({
  // ...
  providers: [CanLoadGuard]
})
export class AppModule {}
```

11.9 小结

在 Web 开发中，路由的概念由来已久，简而言之，就是利用 URL 的唯一性来指定特定的事物，这个事物可以是文件、状态、数据等。服务器端路由早已有之，随着近几年 REST 理念的流行，为更多的人所接受和使用。而浏览器客户端路由，则是随着单页应用（SPA）的兴起，才被越来越多的前端框架所实现的。

本章首先介绍了如何使用 Angular 来开发基本的路由功能，包括路由配置、路由策略、路由跳转、参数传递等；然后进一步深入介绍了子路由、路由拦截及延迟加载等内容。希望通过本章的介绍，能够让读者对路由有一个基本的了解，并能快速地运用到实际项目中。

12

测试

对于一个应用来说，只是完成了基本功能开发，还远远不够，如何在快速迭代中保持稳定的产品质量，测试的重要性不言而喻。从 AngularJS 1.x 到 Angular，Angular 团队都非常重视代码的可测性，并且业界也有不少优秀的测试工具，这使得对通过 Angular 开发的应用，可以游刃有余地编写单元测试及集成测试代码。

本章将介绍测试相关内容，主要包括单元测试和端到端（E2E）测试的概念，测试工具 Jasmine、Karma 和 Protractor 的使用，Angular 自身的测试 API 的讲解，以及 Angular 应用各部分测试方法的介绍。

12.1 概述

测试的意义

毫无疑问，任何应用都可能存在缺陷，软件测试的意义体现在以下几个方面。

- 保证产品质量：测试代码能发现项目中隐藏的缺陷，保证产品质量的稳定，如果缺陷层出不穷，就会很容易让用户失去使用的兴趣，从而失去用户的口碑。
- 提升研发效率：在开发过程中，辅以合理、高效的测试手段，将事半功倍，提升开发效率。例如，可以通过持续集成加自动测试的方式，及时暴露每日新代码的缺陷。代码重构后，也可以迅速进行功能验证。

- 丰富产品说明：对于项目而言，完备的测试代码天然就是优秀的说明文档。通过它即可了解各个功能模块的逻辑关系。

单元测试和端到端测试

对于前端应用来说，最基本的测试无非两种：单元测试和端到端测试。

单元测试，顾名思义，就是测试程序中的某个单元，这里所说的单元往往是一段代码、一个函数等，而非一个大型的功能集合。而端到端测试，可以简单概括为以用户角度来模拟实际操作行为进行的测试，例如模拟用户访问某个网址、单击按钮等。

Angular 测试技术及工具

工欲善其事，必先利其器。

适用于 JavaScript 的测试框架有很多，常见的包括 Mocha、Jasmine、QUnit 等，Angular 团队推荐采用 Jasmine。另外，为了更好地管理测试工程，在多个浏览器上运行测试用例，还需要引入测试过程管理工具，例如 Karma 等。对于端到端测试，则推荐使用 Protractor。下面的章节将主要介绍 Jasmine、Karma 及 Protractor 的基本使用方法。



本章探讨的是如何写好测试代码，也就是自动化测试，而非传统的手工测试。因此，本章的内容基本也是围绕一些测试框架展开的。与手动测试相关的技术，不在本章的讨论范围内。

12.2 单元测试

12.2.1 概述

单元测试是研发过程中必不可少的一个环节，它可以帮助开发人员在更早的阶段（开发阶段甚至是代码设计阶段）发现问题，可以快速、反复地对代码逻辑进行验证。在软件工程中，越早发现问题，意味着修复成本越小。同时，单元测试可以反映出产品本身的功能，开发者可以通过单元测试了解程序的业务逻辑。以下是编写可维护、高质量的单元测试代码的原则。

- 将代码切分成小的可测单元：这要求项目中代码的逻辑和分层都要足够清晰。Angular 从设计之初就追求可测性，按照 Angular 规范编写的代码往往可测性都较好。

- 只测试公开接口：这样可以保证被测代码的变更不会轻易影响到测试的执行。当代码实现细节、私有接口发生改变时，单元测试的运行不受影响。
- 添加到持续集成（CI）中：在持续集成的过程中加入单元测试环节，可以快速主动地反馈单元测试结果，开发人员可以迅速对错误进行相应的处理。

12.2.2 常用测试框架

当测试用例规模逐步增大时，测试工程会面临更多的问题，例如需要将用例进行分组，从而方便管理；或者能够快速运行测试用例，然后以一个友好的界面展示测试的结果和测试覆盖率，等等。在 JavaScript 的世界里，相关的测试框架及工具也非常强大，下面的章节将介绍 Jasmine 和 Karma 相关内容。

12.2.3 Jasmine 介绍

Jasmine 是一个行为驱动的开发测试框架，它不依赖任何其他 JavaScript 框架，也不需要 DOM 操作，它具有灵巧而明确的语法，可以让开发者轻松地编写测试代码。它可运行于服务器端（如 Node.js）和浏览器端。Angular 团队推荐使用 Jasmine 来测试项目代码，甚至基于 Jasmine 扩展了一套测试语法，集成在 @angular/core 包中，后面将具体介绍。

安装 Jasmine

Jasmine 的安装很简单。

- 准备 Node.js 运行环境。
- 创建项目目录，并安装 Jasmine 核心库，命令如下：

```
npm install jasmine-core --save-dev
```



--save-dev 表示将 jasmine-core 添加到 package.json 的 devDependencies 配置中，这个包只在开发时安装。与之对应的 --save，则表示在生产模式下安装，项目在最终发布打包时也要带上这些包。

Jasmine 示例

安装好 jasmine-core 后，会在当前路径下生成子目录 node_modules/jasmine-core。接着新建一个空的 HTML 文件，命名为 test.html。示例代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Jasmine 单元测试</title>
  <link rel="stylesheet" href="node_modules/jasmine-core/lib/jasmine-core/jasmine.
    css">
  <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></script>
  <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine-html.js"></script>
  <script src="node_modules/jasmine-core/lib/jasmine-core/boot.js"></script>
</head>
<body>
  <script>
    // 一段简单的测试代码
    describe("A suite is just a function" , function() {
      var a;
      it("and so is a spec", function() {
        a = true;
        expect(a).toBe(true);
      });
    });
  </script>
</body>
</html>
```

上面展示了一个最简单的单元测试例子，用浏览器打开这个 HTML 文件，页面中的测试代码运行后，将看到该测试用例的执行结果，如图 12-1 所示。

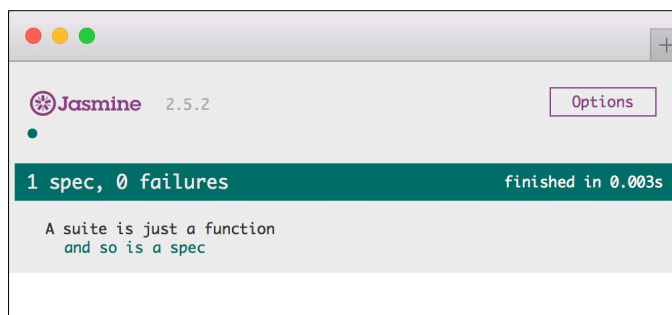


图 12-1 Jasmine 执行测试用例成功

如果把上述例子中的 `a = true` 改成 `a = false`，此时刷新 `test.html` 页面，将显示出错页面，如图 12-2 所示。

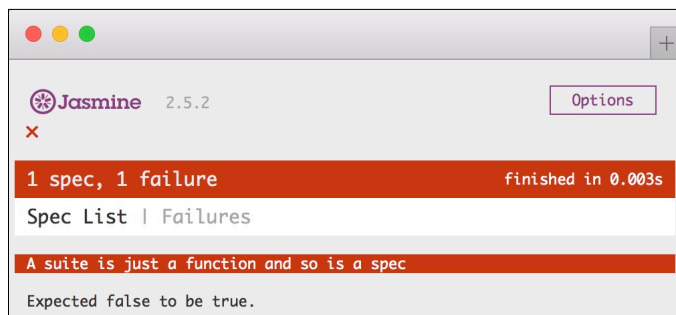


图 12-2 Jasmine 执行测试用例失败

以上代码展示了用 Jasmine 编写单元测试用例的基本步骤。

- 引入文件：在 `<head>` 标签中，需要引入 3 个 JavaScript 文件和 1 个 CSS 文件。
- 设计用例函数：根据测试场景组合使用 `describe()` 和 `it()` 来定义测试用例。
- 编写判定逻辑：在 `it()` 函数中，需要用 `expect()` 断言函数来判定用例测试结果。例如，`expect(a).toBe(true)` 表示变量 `a` 应等于 `true`，否则说明这个用例测试失败。

通过上面几步即可实现用 Jasmine 快速编写测试用例，其中涉及的几个概念，接下来将分别介绍。

测试集

在 Jasmine 中，将功能相似的测试用例统一聚集在测试集（Suite）中，并通过 `describe()` 函数标识。假设被测试的业务类（下面统一简称为被测类）有多个函数，则需要编写多个测试用例，并将这些测试用例都放在同一个 `describe()` 函数中。

测试点

每个具体的功能测试点（Spec）都可以用全局函数 `it()` 来定义，该函数的第一个参数表示该用例的名称，第二个参数则表示测试的细节。`it()` 函数可以包含一个或多个断言（`expect`），每个断言的结果只能是 `true` 或者 `false`，只有当所有断言都为 `true` 时，该测试点才算通过。

内置匹配器

内置匹配器 (Matcher)，就是跟在 `expect()` 函数后面，用于判定结果是否符合期望值的函数。在前面的例子中，已经用到了一个很常用的匹配器 `toBe()`。下面展示常见的匹配器及其用法。

- `toBe()`

`toBe()` 本质是使用操作符 “`===`” 来比较结果值和期望值，`not.toBe()` 则表示不等。示例代码如下：

```
it("The 'toBe' matcher compares with ===" , function() {  
  var a = 15;  
  var b = a;  
  expect(a).toBe(b);  
  expect(a).not.toBe(null);  
});
```

- `toEqual()`

`toEqual()` 用于比较两个对象，可以是自定义的对象类型，也可以是数字、字符串等。示例代码如下：

```
it("Should work for objects" , function() {  
  var foo = {  
    a: 8,  
    b: 15  
  };  
  var bar = {  
    a: 8,  
    b: 15  
  };  
  expect(foo).toEqual(bar);  
});
```

- `toMatch()`

`toMatch()` 用于验证是否匹配正则表达式。示例代码如下：

```
it("The 'toMatch' matcher is for regular expressions", function() {  
  var message = "Angular is good";  
  expect(message).toMatch(/Angular/);  
  expect(message).toMatch("Angular");  
  expect(message).not.toMatch(/bad/);  
});
```

- toContain()

toContain() 用于验证数组是否包含指定元素。示例代码如下：

```
it("The 'toContain' matcher is for finding an item in an Array",function() {
  var a = ["Angular", "React", "jQuery"];
  expect(a).toContain("Angular");
  expect(a).not.toContain("Java");
});
```

为了方便读者查阅，我们将 Jasmine 提供的匹配器加以整理，如表 12-1 所示。

表 12-1 Jasmine 内置匹配器

匹配器	作用
toBe	使用 “===” 比较结果
toEqual	比较两个对象是否相等
toMatch	正则表达式匹配
toBeNull	验证是否为 null
toBeTruthy	验证是否为 true
toBeFalsy	验证是否为 false
toBeLessThan	验证结果是否小于指定值
toBeGreaterThan	验证结果是否大于指定值
toContain	验证数组是否包含指定元素
toBeCloseTo	将值进行四舍五入后比较是否相等，如 toBeCloseTo(e, 2)，数字 2 表示数字精度
toThrow	验证函数是否抛出一个错误
toThrowError	验证函数是否抛出指定的错误
toBeDefined	验证对象是否不为 undefined

beforeEach、afterEach 函数

另外，在测试中还会经常用到 beforeEach()、afterEach() 这两个函数，用于定义每个测试用例执行前及执行后的公共逻辑。

- beforeEach(): 定义了每个用例在执行前，所需要执行的初始化函数。
- afterEach(): 每个用例在执行结束后，均将执行 afterEach() 函数。

12.2.4 Karma 介绍

在 Angular 测试中，可以用 Karma 自动化管理由 Jasmine 编写的单元测试用例，两者相互配合，使得单元测试可以更高效地进行。Karma 是一个基于 Node.js 的测试执行过程管理工具（Test Runner），其前身是 Google 在 2012 年开源的 Testacular。它主要带来了以下这些强大的特性。

- 在真实浏览器上测试：前端测试的一个目标，就是关注代码在不同浏览器上的表现，检查是否有兼容性问题。Karma 可以同时启动 Chrome、Safari 等不同的浏览器，并在一个浏览器窗口展示各个浏览器的测试结果。
- 自动触发测试：在开发过程中，如果被测代码或者测试代码发生了变更，就能自动触发测试动作，新代码能马上得到自动验证，而且整个过程不用在多个编辑器、浏览器窗口之间切换，全由后台自动执行。
- 丰富的插件支持：借助于 Karma 的各类插件，可以很方便地扩展功能，如生成友好的覆盖率测试报告、支持 Webpack 和 SystemJS 打包工具等。

对于只有几个测试用例的小型示例工程，Karma 也许显得大材小用，但随着项目规模逐渐变大，一款值得信赖而强大的测试管理工具，将为你解决编写测试代码之外的很多烦心事。

12.2.5 Karma 结合 Jasmine 测试

为了帮助读者快速上手 Karma 及 Jasmine 的测试开发，这里将展示一个简单的例子。

安装 Karma

在系统中全局安装 Karma，这样才能在命令行窗口中执行 Karma 命令：

```
npm install -g karma-cli  
npm install -g karma  
npm install -g karma-jasmine karma-chrome-launcher
```



本章后续会讲解如何结合 Jasmine 进行测试，因此需要安装 Jasmine 相关插件。

初始化配置文件

在项目根目录下执行 `karma init` 命令，将启动 Karma 的配置生成器，生成器将依次询问要使用的配置，此时可以一直按回车键，使用默认配置即可，最终将在本地生成 `karma.conf.js` 配置文件。Karma 配置生成界面如图 12-3 所示。

图 12-3 Karma 配置生成界面

安装依赖包

在根目录下创建 `package.json`，然后填入以下配置：

```
{
  "name": "test",
  "scripts": {
    "test": "karma start karma.conf.js"
  },
  "devDependencies": {
    "jasmine": "^2.4.1",
    "jasmine-core": "^2.4.1",
    "karma": "^1.1.0",
    "karma-chrome-launcher": "^1.0.1",
    "karma-jasmine": "^1.0.2"
  }
}
```

这是一个简单的配置文件，用于使用 npm 管理项目和配置依赖。其中：

- "test": "karma start karma.conf.js" 表示为 karma start karma.conf.js 命令创建了一个别名 npm test，运行 npm test 相当于运行 karma start karma.conf.js。
- devDependencies 配置项代表设置项目开发时所用的依赖包，上例中该配置项引入的是 Karma 及 Jasmine。

然后执行 npm install 命令来安装 package.json 中指定的依赖包。



前面章节中已经介绍过 package.json，每一个使用 npm 来管理依赖包的工程，都会用到这个配置文件。

编写测试用例

在根目录下创建一个名为 test.js 的代码文件，它包含了一个简单的函数，用于验证参数是否为 string 类型。示例代码如下：

```
function isString(s){  
    return typeof(s) === "string";  
}
```

在同级目录下创建一个测试用例文件，命名为 test.spec.js，然后输入以下代码：

```
describe("A suite of string util" , function() {  
    it("is string", function(){  
        expect(isString("ss")).toBeTruthy();  
        expect(isString(1)).toBeFalsy();  
    });  
});
```

配置测试代码路径

在 karma.conf.js 中配置测试及被测代码的路径，这里可以用通配符的方式，新添加的配置项如下：

```
files: [  
    '*.js'  
],
```

修改后的 karma.conf.js 配置内容如下：

```
module.exports = function(config) {  
  config.set({  
    basePath: '',  
    frameworks: ['jasmine'],  
    files: ['*.js'],  
    exclude: [],  
    preprocessors: {},  
    reporters: ['progress'],  
    port: 9876,  
    colors: true,  
    logLevel: config.LOG_INFO,  
    autoWatch: true,  
    browsers: ['Chrome'],  
    singleRun: false,  
    concurrency: Infinity  
  })  
}
```

在上面的 Karma 配置中，需要关注的配置项如下。

- **frameworks**：指定使用何种测试框架，如上述配置中，则指定了使用 Jasmine 作为测试框架。
- **browsers**：表示在何种浏览器上运行测试用例，可以指定 Chrome、Safari 等浏览器。
- **files**：指定测试和被测代码的路径。因为本例中的测试及被测代码文件名后缀均为 “.js”，且都放在根目录下，因此设置 files 配置项后，Karma 将引入这些.js 文件，并执行其中的测试用例。另外，basePath 和 exclude 也是文件路径的相关配置，其中 basePath 定义了 files 配置的基础文件路径。basePath 配合 files 使用相当于指定了 Karma 查找的完整文件路径，而 exclude 配置项则为不纳入测试范围的文件路径。
- **preprocessors** 和 **reporters**：用于配置插件。Karma 可以在执行测试及输出报告时引入其他插件，例如在测试运行前执行 Webpack 编译，在输出结果时输出代码覆盖率报告等，这些丰富的功能都来自插件的扩展。
- **autoWatch**：表示当修改了测试或者被测试的文件时，是否重新执行测试用例。

上面未提到的配置项，如 port、colors、logLevel、singleRun、concurrency 等，感兴趣的读者可以在 Karma 官网进行了解。

运行测试用例

最后，执行 `npm test` 命令，Karma 将自动打开一个 Chrome 浏览器窗口，显示的界面如图 12-4 所示。



图 12-4 浏览器运行 Karma 的效果

在命令行窗口中将按执行顺序，依次显示如图 12-5 所示的信息。注意在最后一行，可以看到一个 SUCCESS 的结果，表示成功执行了一个用例。

```
angular — Google Chrome Helper * npm NVM_RC_VERSION= NVM_IOJS_ORG_VERSION_LISTING=http...
└─ npm test

> test@ test /private/tmp/angular
> karma start karma.conf.js

21 11 2016 09:35:27.625:WARN [karma]: No captured browser, open http://localhost:9876/
21 11 2016 09:35:27.639:INFO [karma]: Karma v1.3.0 server started at http://localhost:9876/
21 11 2016 09:35:27.639:INFO [launcher]: Launching browser Chrome with unlimited concurrency
21 11 2016 09:35:27.646:INFO [launcher]: Starting browser Chrome
21 11 2016 09:35:30.461:INFO [Chrome 54.0.2840 (Mac OS X 10.11.6)]: Connected on socket /#g8-3TgerkGg
q6_FbAAAA with id 74766364
Chrome 54.0.2840 (Mac OS X 10.11.6): Executed 1 of 1 SUCCESS (0.014 secs / 0.002 secs)
```

图 12-5 Karma 执行结果

综上所述，配置 Karma 和 Jasmine 的组合，只需要以下三个重要步骤。

- 首先，安装 Karma 并通过 `karma init` 命令初始化 Karma 配置文件。
- 然后，往 `package.json` 中添加 Karma 及 Jasmine 依赖。
- 最后，用 Jasmine 语法编写测试用例。

为了便于读者理解 Karma 和 Jasmine 的基本使用方式，上文中特意未掺杂 Angular API，在了解了这两个工具后，下面我们开始学习 Angular 应用场景下的测试。

12.3 Angular 单元测试

12.3.1 概述

在前文中，已经提到 Angular 特别注重应用的可测性。Angular 中的组件化、依赖注入等设计理念，使其进行单元测试更加便捷。对 Angular 应用进行单元测试，主要涉及以下知识点：

- Angular 应用下的组件、服务、管道等，都是普通的 TypeScript 类，因此可以在测试代码中初始化被测类，然后调用被测类中的方法，验证这些方法的返回值。我们把这种方式称之为独立单元测试（Isolated Unit Test）。
- Angular 框架在 `@angular/core/testing` 目录下提供了不少测试工具，这些与测试相关的 API 可以用来创建 Angular 的测试环境和执行相关任务（如触发变化监测），设置 Providers 和调用注入器（Injector）来实现依赖注入机制，与页面中的 DOM 元素交互并模拟用户行为，以及验证服务中异步请求的耗时情况等。
- 测试用例代码不是独立的，它们需要和项目的前端打包构建工具结合。项目可能用了 Webpack 或 SystemJS 等构建工具，对于不同的打包构建工具，对应的 Karma 配置也有所不同。良好的配置有助于构建一个完整的、可持续维护的测试工程。

掌握 Angular 应用单元测试需具备的知识如图 12-6 所示。

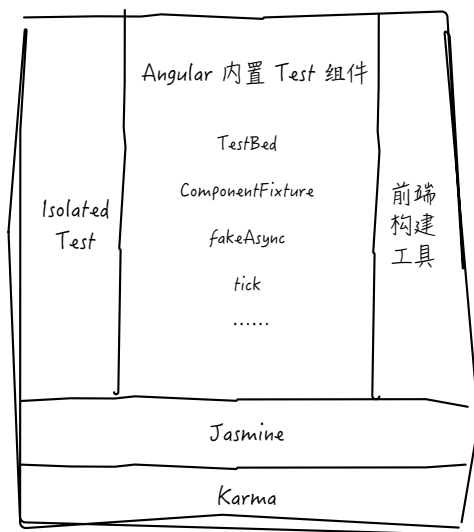


图 12-6 Angular 测试需掌握的知识体系

为了更好地讲解 Angular 测试的知识点，本节将围绕通讯录例子的联系人详情页进行测试，该例子通过 `ContactService` 获取源数据，然后由 `DetailComponent` 组件负责渲染数据到对应的模板中。联系人详情页面如图 12-7 所示。



图 12-7 联系人详情页面

下面首先会讲述测试环境的搭建，然后基于这个环境尝试一个简单独立的单元测试，最好重点介绍 Angular 的测试工具集，并逐步介绍如何测试一个 Angular 应用。

通讯录例子采用 Angular CLI 创建，Angular CLI 深度整合了 Karma 测试工具，在项目创建完后测试工具也已经安装并且配置好。Angular CLI 创建好的 `karma.config.js` 文件配置如下：

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular/cli'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage-istanbul-reporter'),
      require('@angular/cli/plugins/karma')
    ],
  });
};
```

```
client:{
  clearContext: false // leave Jasmine Spec Runner output visible in browser
},
files: [
  { pattern: './src/test.ts', watched: false }
],
preprocessors: {
  './src/test.ts': ['@angular/cli']
},
mime: {
  'text/x-typescript': ['ts','tsx']
},
coverageIstanbulReporter: {
  reports: [ 'html', 'lcovonly' ],
  fixWebpackSourcePaths: true
},
angularCli: {
  environment: 'dev'
},
reporters: config.angularCli && config.angularCli.codeCoverage
           ? ['progress', 'coverage-istanbul']
           : ['progress', 'kjhtml'],
port: 9876,
colors: true,
logLevel: config.LOG_INFO,
autoWatch: true,
browsers: ['Chrome'],
singleRun: false
});
};
```

由于 Angular CLI 集成了 Karma 测试，在构建项目或者创建组件时就可以自动生成测试文件了，生成规则如下。

- 测试文件的路径：被测文件和测试文件放在同一个层级目录中。
- 测试文件名：保持和被测文件相同的文件名前缀，并加上 spec 标识其为测试文件。假设被测文件名为 `detail.component.ts`，则对应的测试用例文件则命名为 `detail.component.spec.ts`。



`detail.component.spec.ts` 可以使用 Angular CLI 的命令行 `ng generate component detail` 自动生成。关于 Angular CLI 的命令将在本书第三部分详解。

12.3.2 独立单元测试

先来看看，如果不依赖 `@angular/core/testing` 这个套件，按照通用的方法来写测试代码，能达到什么样的效果。

组件的测试

这里将测试一个简化版（跟通讯录例子的代码不一样，这里只是为了方便举例说明）的联系人详情组件。示例代码如下：

```
// detail.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'detail',
  templateUrl: 'app/src/detail/detail.component.html',
  styleUrls: ['app/src/detail/detail.component.css'],
})
export class DetailComponent {

  detail:any = {};

  getById(id:number) {
    this.detail = {
      "id": 1,
      "name": "张三",
      "telNum": "18900001001",
      "address": "广东省深圳市",
      "email": "123@qq.com",
      "birthday": "1990/10/10",
      "collection": 1
    };
  }
}
```


它对应的组件模板示例代码如下：

```
<!-- detail.component.html -->
<div class="detail-contain">
  <ul class="detail-info">
    <li id="test">
      <p> 手机号码: </p>
      <p>{{ detail.telNum }}</p>
    </li>
    <li>
      <p> 邮箱: </p>
      <p>{{ detail.email }}</p>
    </li>
    <li>
      <p> 生日: </p>
      <p>{{ detail.birthday }}</p>
    </li>
    <li>
      <p> 住址: </p>
      <p>{{ detail.address }}</p>
    </li>
  </ul>
</div>
```

可以看到，上面例子中没有包含路由、服务、管道等其他模块，这意味着还不用考虑依赖注入、响应耗时及 Mock 服务等问题。在 DetailComponent 组件代码中，通过 `getById()` 方法，将固定不变的数据赋值给变量 `detail` 并绑定到模板中，从而实现了页面的渲染。

组件是 Angular 应用中最重要的重要组成部分。每一个组件本质上都是 TypeScript 类，因此，测试组件跟测试普通的 TypeScript 类没什么不同。

对应到这个例子，先在 `beforeEach()` 方法中初始化 DetailComponent 组件类，然后验证类的成员变量 `detail` 是否等于预期值。示例代码如下：

```
// detail.component.spec.ts
import { DetailComponent } from './detail.component';

describe('detail.component', () => {

  // 步骤 1
```

```

beforeEach(() => {
  this.component = new DetailComponent();
});

// 步骤 2
it('test simple component', () => {
  this.component.getById(1);
  expect(this.component.detail.name).toEqual("张三");
});
});

```



在上面的 Jasmine 小节中介绍过 `beforeEach()` 函数，它的作用是在执行每个测试用例前，可以预处理一些必要逻辑。

如果被测类中新增一个函数，也可以采用同样的测试方式，例如新增了 `getTelCity()` 函数，那么对应的测试示例代码如下：

```

// ...
expect(this.component.getTelCity()).toBe('xxx');
// ...

```

管道的测试

假设联系人详情组件中的电话号码需要进行特殊的格式化处理，此时可以引入管道来解决。与组件一样，管道本质上也是一个 TypeScript 类，所以测试管道的方法，跟测试普通的类也没有差别。电话号码格式化管道的示例代码如下：

```

// phone.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'phone'
})
export class PhonePipe implements PipeTransform {
  transform(val:string): string {
    if(!val) return '';
    if(val.length === 11) {
      return val.replace(/(\d{3})(\d{4})(\d{4})/, (m, m1, m2, m3) => {

```

```

        return [m1, m2, m3].join('-');
    })
}
return val;
}
}

```

对应的测试代码如下:

```

// phone.pipe.spec.ts
import { PhonePipe } from './phone.pipe';

describe('test PhonePipe', () => {
    beforeEach(()=> {
        this.pipe = new PhonePipe();
    });
    it('transforms phone', () => {
        expect(this.pipe.transform('13566666666')).toEqual('135-6666-6666');
    });
});

```

服务的测试

Angular 服务的测试跟组件的测试基本一致。假设有一个 `SomeNameService` 服务, 里面有一个 `getNameById()` 方法, 在 `some-name.service.spec.ts` 测试文件中对其进行测试。示例代码如下:

```

// some-name.service.spec.ts
import { SomeNameService } from './some-name.service';
beforeEach(()=> {
    this.service = new SomeNameService();
});
it('test getNameById()', () => {
    expect(this.service.getNameById(1)).toBe("xxx");
});

```

如果该服务依赖其他服务, 则需要对所依赖的服务也进行初始化。如 `SomeNameService` 依赖 `OtherService`, 则对应的示例代码如下:

```

@Injectable()
export class SomeNameService {
    constructor(private otherService: OtherService) { }
}

```

```
  getNameById() { return this.otherService.getValue(); }  
}
```

这种情况需要先后初始化两个服务。示例代码如下：

```
beforeEach(()=> {  
  const otherService = new OtherService();  
  this.service = new SomeNameService(otherService);  
});
```

也可以使用简单的对象模拟服务实例，这样可以防止无尽的依赖链初始化，专注于被测服务。示例代码如下：

```
beforeEach(()=> {  
  const fakeOtherService = {  
    getValue: () => '模拟数据'  
  }  
  this.service = new SomeNameService(fakeOtherService as OtherService);  
});
```

12.3.3 测试工具集

服务和管道比较独立，使用基本的测试方法也能很好地覆盖所有功能。但涉及组件的测试，上述例子只能覆盖组件类的基本功能，诸如异步请求、模板测试等其他很常见的功能，用独立的方法测试是非常困难的。所以如果需要真正地深入 Angular 测试，不得不借助于 @angular/core/testing 测试套件提供的辅助方法和工具类。

使用 TestBed

相比上述使用简单的 new 方式初始化被测类，@angular/core/testing 测试套件提供了 TestBed 工具集来初始化类。例如上面 detail.component.spec.ts 的测试代码和下面的测试代码作用是一样的：

```
// detail.component.spec.ts  
import { async, ComponentFixture, TestBed } from '@angular/core/testing';  
describe('DetailComponent', () => {  
  let component: DetailComponent;  
  let fixture: ComponentFixture<DetailComponent>;  
  
  beforeEach(async(() => {  
    TestBed.configureTestingModule({  
      declarations: [  

```

```
        DetailComponent
      ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(DetailComponent);
    component = fixture.componentInstance; // 获取组件实例
  });

  it('test simple component', () => {
    component.getById(1);
    expect(component.detail.name).toEqual("张三");
  });
});
```



`component.getById(1)` 获取数据这一步很多测试用例都需要使用到，可以放到 `beforeEach` 阶段。

上面的代码首先使用 `TestBed.configureTestingModule()` 构建组件测试依赖的模块环境，该方法の入参对象跟 `@NgModule` 的元数据对象几乎一样。注意到这里使用 `async()` 函数对模块构建进行包装，主要是因为 `DetailComponent` 依赖外部的模板和样式文件，需要有一个读取的过程才能进行编译。所以这是一个异步操作，使用 `async()` 包装可以保证第一个 `beforeEach()` 引入的异步回调任务先完成，再执行第二个 `beforeEach()` 引入的同步任务。

接下来，在第二个 `beforeEach()` 里使用 `TestBed.createComponent()` 初始化被测组件，并创建 `ComponentFixture` 对象，这个对象可以认为是被测组件的上下文环境，通过它可以获取已经完成初始化的组件实例及 DOM 元素等。



使用两个 `beforeEach()` 并不是必需的，由于 `compileComponents()` 函数返回的是 `promise` 对象，所以可以把组件实例创建逻辑放到 `compileComponents().then()` 函数里完成。不过这种方式的可读性不是太好，所以更推荐使用两个 `beforeEach()`。

拿到了组件实例即可对它的方法进行测试，如 `getById()` 方法。

组件 DOM 交互测试

上面只是介绍了测试组件类的方式，更多复杂的测试需求，例如测试 HTML 输出，或者组件 DOM 交互逻辑等，都可以借助于 TestBed 来实现。

Angular 提供了与 TestBed 相关的一系列测试 API，用于帮助开发者构建测试上下文环境。通过它提供的 API，我们可以很方便地完成诸如依赖注入、初始化被测组件及获取组件 DOM 元素等测试任务。

通过 TestBed 进行组件的初始化，可以获取到组件模板对应的 DOM 元素，以及检查 DOM 元素的内容等，而这些仅仅使用 `new` 操作实例化组件是很难做到的。

在上面基于 TestBed 的代码里添加新的测试用例。示例代码如下：

```
it('test simple component with TestComponentBuilder', () => {  
  
  // 这一步不能省略，变化监测执行完成后数据才能更新到 DOM 元素上  
  fixture.detectChanges();  
  
  const el: HTMLElement = fixture.debugElement.nativeElement;  
  
  // 获取并检查 DOM 元素的内容  
  const tplValue = el.querySelector('.detail-info>li:first-child>p:nth-child(2)').  
    textContent;  
  expect(tplValue).toBe('18900001001');  
});
```

在上面的代码中，需要关注以下知识点：

- `fixture.detectChanges()` 方法用于当 JavaScript 变量及模板内容有变更时手动触发变化监测。当组件初始化、DOM 元素或 JavaScript 值变化时，都需要调用这个方法，以便触发 Angular 的变化监测机制。
- `fixture.debugElement.nativeElement` 用于获取组件对应的原生 DOM 元素，之后可以通过 `querySelector()` 等 DOM 元素原生 API 做进一步处理。

组件测试的更多技巧

触发点击事件

如 `fixture.debugElement.nativeElement.querySelector('button').click()`, 将获取 `button` 子元素, 并触发 `click` 事件。

替换模板

在初始化组件时, 可以根据测试需求, 通过 `TestBed.overrideComponent()` 方法将组件模板替换为其他模板。示例代码如下:

```
TestBed.overrideComponent(DetailComponent, {
  set: {
    template: ` < span > Foo < span > `
  }
}).compileComponents().then(() => {
  // 测试逻辑
});
```

自定义匹配器

在上面的例子中, `el.querySelector('.detail-info>li:first-child>p:nth-child(2)` 这个选择器写得非常烦琐, 如果我们想更简单地在整个组件模板的 HTML 范围内检索是否存在某个值, 应该怎么做呢?

可以通过 `DOM.textContent.indexOf()` 进行检索。为了方便复用, 可以通过 Jasmine 的 `addMatchers()` 方法进行抽象, 这样以后其他测试用例也可以方便地使用这个匹配器。示例代码如下:

```
// ...
beforeEach(() => {
  jasmine.addMatchers({
    toContainText: function() {
      return {
        compare: function(actual : any, expectedText : string) {
          var actualText = actual.textContent;
          return {
            pass: actualText.indexOf(expectedText) > -1,
            get message() {
              return 'Expected' + actualText + 'to contain' + expectedText;
            }
          }
        }
      }
    }
  });
});
```

```

        };
    }
    };
}
});
});

```

定义了这个匹配器之后，后续在组件测试中就可以直接使用了。示例代码如下：

```

// ...
var el = fixture.debugElement.nativeElement;
component.getById(1);
fixture.detectChanges();
expect(el).toContainText('18900001001'); // 扩展 addMatchers

```

通过依赖注入测试

在实际开发中，组件往往还会依赖服务，接下来将改造之前的 DetailComponent 组件，使其通过服务获取组件类的成员变量 detail 的信息。示例代码如下：

```

// ...
export class DetailComponent {
  detail:any = {};
  constructor(
    private contactService: ContactService // 注入组件需要的服务类
  ) { }

  getById(id: number) {
    this.detail = this.contactService.getContactById(id);
  }
}

```

此时，测试代码也要做相应的更新，通过 providers 配置项注入 ContactService（其他地方保持不变）。示例代码如下：

```

// import ContactService
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [
      DetailComponent
    ],
    providers: [ ContactService ] // 注入 ContactService
  });
});

```



```
    })  
    .compileComponents();  
  }));
```

这种方式虽然可行，但是把 `ContactService` 也牵涉到组件测试里了，解决方法是通过 Mock 方式测试。

通过 Mock 方式测试

在单元测试中，经常会提到 Mock。所谓 Mock，就是构造一个虚拟对象（或数据），代替真实的测试对象（或数据），以方便对测试代码中真正关注的对象进行测试。例如，在 `DetailComponent` 组件中调用了 `ContactService` 服务，在对 `DetailComponent` 组件进行单元测试时，我们关心的是 `DetailComponent` 组件本身的逻辑，而 `ContactService` 服务的逻辑不应该影响测试逻辑的编写。假如在 `ContactService` 服务中需要进行 HTTP 调用，那对于测试结果及耗时都是不可控的。因此，我们希望可以方便地模拟 `ContactService` 服务各接口的返回值，通过 Mock 方式就可以达到这个目的，而 Mock 之后对于 `DetailComponent` 是透明的，不需要任何修改，如图 12-8 所示。

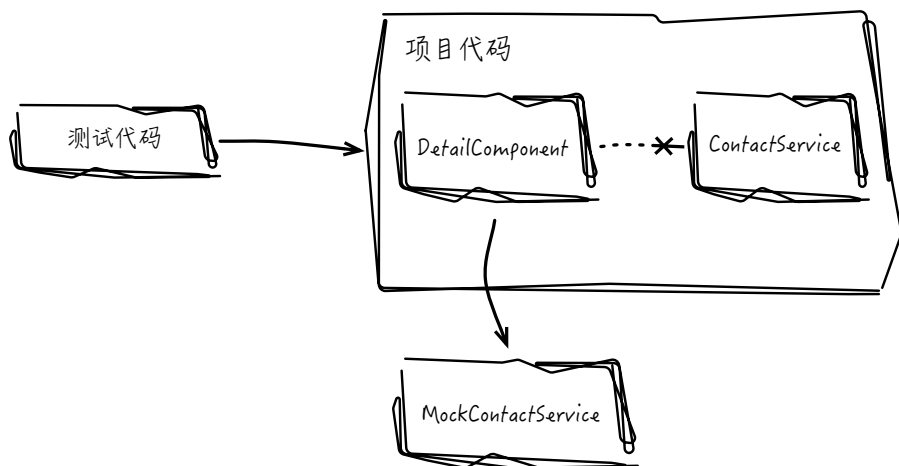


图 12-8 Mock 服务示意图

首先，在测试代码中写一个 Mock 服务，并按照测试需求，重写 `ContactService` 服务中的 `getContactById()` 方法。示例代码如下：

```
class MockContactService extends ContactService {  
  getContactById(id: number) {
```

```
    return {  
      "name": "张三"  
    };  
  }  
}
```

然后，声明使用 MockContactService 服务。示例代码如下：

```
beforeEach(async(() => {  
  TestBed.configureTestingModule({  
    declarations: [  
      DetailComponent  
    ],  
    providers: [ provide: ContactService, useClass: MockContactService ] // 替换  
                ContactService  
  })  
  .compileComponents();  
}));
```

这样在执行测试代码时，实际上 Angular 会注入 MockContactService 类来替换 ContactService 服务，并调用 MockContactService 服务的 getContactById() 方法，返回模拟数据并测试相应的逻辑。

除创建一个新的服务类来替换原有的服务外，还有更简单的方式，即直接创建一个对象来替换服务。示例代码如下：

```
// import ContactService  
  
const ContactServiceStub = {  
  getContactById(id: number) {  
    return {  
      "name": "张三"  
    };  
  }  
}  
  
beforeEach(async(() => {  
  TestBed.configureTestingModule({  
    declarations: [  
      DetailComponent  
    ],  

```

```

        providers: [ provide: ContactService, useValue: ContactServiceStub ] // 替换
                      ContactService
    })
    .compileComponents();
  }));

```

测试异步请求

很多组件都需要请求远端服务器拉取数据，这是典型的异步操作，而这些异步操作通常封装在某个服务下。要测试这样的组件，除可以使用上述提到的 Mock 方式之外，还可以使用 spy 方式模拟服务函数的执行，来达到模拟异步数据返回的效果（并不真正地发出请求）。

把 DetailComponent 改为异步拉取数据。示例代码如下：

```

// ...
export class DetailComponent {
  detail:any = {};
  constructor(
    private contactService: ContactService // 注入组件需要的服务类
  ) { }

  getById(id: number) {
    // 异步拉取数据
    this.contactService.getContactById(id).subscribe(detail => {
      this.detail = detail;
    })
  }
}

```

运行测试用例会发现，原先的测试用例会报错。DetailComponent 的测试并不关心 contactService.getContactById() 函数的具体实现，组件只关心该函数返回的是否是 Observable 类型的数据。所以可以使用 spy 方式模拟返回的数据，对测试代码改造如下：

```

beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [
      DetailComponent
    ],
    providers: [ ContactService ] // 真正的ContactService
  })
})

```

```
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(DetailComponent);
    component = fixture.componentInstance; // 获取组件实例

    // 从注入器里获取 ContactService 实例
    let contactService = fixture.debugElement.injector.get(ContactService);

    // 利用 spyOn 处理 getContactById() 函数
    spy = spyOn(contactService, 'getContactById')
      .and.returnValue(Observable.of({ "name": "张三" }));
  });
```

构建完 spy 对象后，接下来就可以写测试用例了。示例代码如下：

```
it('should get detail value', () => {
  component.getById(1);
  expect(component.detail.name).toEqual('张三');
  expect(spy.calls.any()).toEqual(true);
});
```

12.4 端到端测试

12.4.1 概述

端到端测试（End to End Test，有时也简称为 E2E Test），从另一个角度理解，就是模拟用户行为来进行测试。例如，模拟打开某个网页，单击指定按钮；是否按预期弹出窗口或跳转页面；提交 form 表单后，是否弹框提示正确的信息等。端到端测试还有一个特点，就是测试代码无须依赖被测代码，对于 Web 应用，端到端测试就是直接测试最终的产品，即网页。由于它是模拟用户行为的一种测试，编写各种测试用例往往比较耗时，测试效率一般也会低于单元测试。也正因如此，在大多数情况下大家会倾向于单元测试，而非端到端测试。

尽管如此，为了丰富及补充测试的手段，开发者仍需要了解如何进行端到端测试，并在合适的场景下使用。

接下来，将介绍如何使用 Protractor 测试工具搭建端到端的测试环境，以及编写测

试用例。

12.4.2 Protractor 介绍

Protractor 是一个专门为 Angular 设计的端到端测试框架。Protractor 提供了一系列 API，帮助开发者编写端到端的测试用例，它可以启动一个真实的浏览器进行测试。Protractor 默认使用 Jasmine 测试框架，所以掌握了 Jasmine 的基本语法后，只需再学习 Protractor 提供的少数 API，就可以很方便地编写端到端的测试用例。

Angular CLI 已经深度整合了 Protractor，CLI 在创建项目时就已安装好相关的依赖包，并且初始化好 Protractor 的配置文件。示例配置如下：

```
const { SpecReporter } = require('jasmine-spec-reporter');

exports.config = {
  allScriptsTimeout: 11000,
  specs: [
    './e2e/**/*.e2e-spec.ts'
  ],
  capabilities: {
    'browserName': 'chrome'
  },
  directConnect: true,
  baseUrl: 'http://localhost:4200/',
  framework: 'jasmine',
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 30000,
    print: function() {}
  },
  beforeLaunch: function() {
    require('ts-node').register({
      project: 'e2e/tsconfig.e2e.json'
    });
  },
  onPrepare() {
    jasmine.getEnv().addReporter(new SpecReporter({ spec: { displayStacktrace: true } }));
  }
};
```

在上述配置中，指定了 Jasmine 作为测试框架，并通过 specs 选项配置了测试用例文件的路径（在 e2e 文件夹下以 e2e-spec.ts 结尾的文件），接下来就可以开始编写测试用例了。

编写测试用例

在 e2e 文件夹下创建一个测试用例代码文件，用于模拟访问通讯录例子中的联系人列表页面，命名为 list.e2e-spec.ts。测试代码模拟单击第一个条目，前往对应的联系人详情页。示例代码如下：

```
import { browser, element, by } from 'protractor';
describe('contact list', function() {
  it('test ListComponent', function() {
    // 打开网页，在 Protractor 配置里已经指定了 baseUrl，只写路径即可
    browser.get('/list');

    const contactList = element.all(by.css('.list li a'));

    // 测试列表记录条数是否符合预期
    expect(contactList.count()).toBeGreaterThan(8);

    contactList.first().click();
    browser.getCurrentUrl().then(function(url) {
      expect(url.endsWith('list/1')).toBe(true);
    });
  });
});
```

上述代码中的 describe、it、expect 等这些语法来自于 Jasmine 测试框架。Protractor 则提供了以下比较常用的 API。

- 打开网页：如 browser.get('/list')。
- 元素选择：element.all(by.css('.list li a')) 将返回页面上满足 CSS 规则的元素。
- 交互动作：获取元素之后，可以通过 sendKeys() 往 input 文本框中输入内容，可以通过 click() 单击元素，也可以通过 getText() 获取元素等。



关于更多 API 的用法, 请参考 Protractor 官网 (<http://www.protractortest.org/#/api>), 这里不再赘述。

运行测试用例

运行下面命令即可启动测试:

```
ng e2e
```

运行该命令之后, CLI 开始构建打包项目代码。构建完成后, 默认打开 Chrome 浏览器, 然后跳转到测试用例中指定的网址, 执行对应的测试动作并验证测试结果。测试用例运行完成后, 命令行窗口显示的内容大致如下:

```
[11:25:00] I/update - chromedriver: file exists /Users/xxx/angular-contacts-demo/
node_modules/protractor/node_modules/webdriver-manager/selenium/chromedriver_2
.31.zip
[11:25:00] I/update - chromedriver: unzipping chromedriver_2.31.zip
[11:25:01] I/update - chromedriver: setting permissions to 0755 for /Users/xxx/
angular-contacts-demo/node_modules/protractor/node_modules/webdriver-manager/
selenium/chromedriver_2.31
[11:25:01] I/update - chromedriver: chromedriver_2.31 up to date
[11:25:01] I/launcher - Running 1 instances of WebDriver
[11:25:01] I/direct - Using ChromeDriver directly...
Spec started

    contact list
      ✓ test ListComponent

Executed 1 of 1 spec SUCCESS in 1 sec.
[11:25:04] I/launcher - 0 instance(s) of WebDriver still running
[11:25:04] I/launcher - chrome #01 passed
```

由上面的运行结果可以看出, 我们运行了 1 个测试用例, 0 个失败, 表明测试用例执行成功。

12.5 小结

在本章中，首先介绍了单元测试、端到端测试的基本概念；然后简单介绍了 Jasmine 及 Karma 的使用方式；接着结合 Angular 的实际测试场景，分别介绍了组件、服务和管道的测试方法及相关测试用例的实践；最后介绍了端到端测试及对应的 Protractor 框架。

本章内容仅仅是测试的基础入门知识，更多的测试技巧，还需要结合项目的实际情况逐步摸索。当阅读到这里时，我们就已经完成了本书第二部分全部内容的学习，对开发 Angular 应用所需的知识点如组件、模板、指令、服务与 RxJS、依赖注入和路由等概念应该有了基本的认识，同时也应该能运用这些知识对 Angular 应用编写简单的单元测试及端到端测试用例了。接下来，让我们进入实战部分，将在前面章节中学到的一些知识运用到实际项目中，做到真正的学以致用！

第三部分

实战篇

- 问卷调查系统简介
- 项目起步
- 问卷编辑模块
- 我的问卷模块
- 用户管理模块
- 项目构建和最佳实践

13

问卷调查系统简介

通过前面两大部分内容的学习，相信读者已经对 Angular 的核心模块及架构理念有了一定的认识和理解，接下来我们将学习如何把这些 Angular 相关知识整合起来，并构建一个完整的 Web 应用。

在本部分的章节中，将会使用 Angular 来实现一个简单的问卷调查系统。该问卷调查系统将会使用到前面介绍过的大部分知识，如组件、指令、服务和路由等，相信读者在完成本部分内容的阅读及实践后，将会更深刻地理解 Angular 各核心模块是如何协同工作的。

本章将介绍问卷调查系统的行业背景，并详细分析该系统的具体功能特性，后面几个章节的内容将会围绕这些功能特性的具体实现来展开介绍。通过本章的阅读，相信读者对整个实战项目应该能有个全局的认识。

13.1 项目背景

在大数据技术高速发展的今天，有效的数据采集是所有大数据产品的基础，也是关键的一环。数据采集的方式多种多样，比如电商网站可以通过日志系统采集用户的访问历史信息，进而通过大数据平台分析和数据挖掘来发现用户感兴趣的商品种类，并有针对性地进行产品推广。再比如在我们经常使用的地图类应用中，可以通过采集正在使用的地理位置信息来分析特定区域的交通拥堵情况，进而给出合理的出行线路的建议。而

问卷调查是一种更直接、更有针对性的数据采集手段。

目前国内已经涌现出了问卷网、问卷星、腾讯问卷及调查派等知名的在线问卷调查网站。接下来我们要实现的问卷调查系统，参考了目前较为流行的问卷调查网站的实现方式，并对某些功能做了简化处理。这样一方面降低了示例的复杂度，可以较为全面地涵盖 Angular 的各个核心概念，另一方面也确保了示例代码的准确性和可读性。

13.2 主要特性

在标准的软件开发流程中，一般需要先进行需求分析。在需求分析阶段，要明确系统试图解决的问题、解决问题的方式，以及系统提供的操作界面和用户完成具体需求的操作路径等其他细节，这个步骤通常是由专门的产品经理来完成的。接下来先简单地分解问卷调查系统应有的模块和功能，如图 13-1 所示。

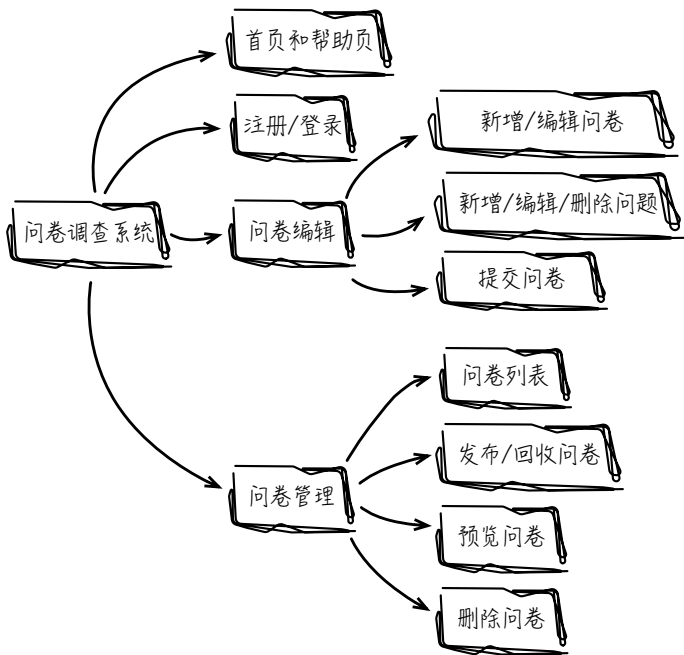


图 13-1 产品模块划分

为了帮助读者更好地了解我们将要实现的问卷调查系统，首先来看系统中涉及的所有页面，如图 13-2 所示。

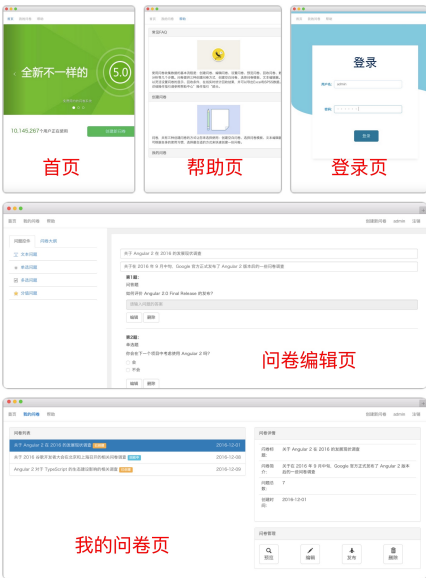


图 13-2 主要产品页面

13.2.1 首页和帮助页

问卷调查系统的首页提供了系统的各项功能入口和简单的介绍，而帮助页则提供了系统各个功能模块的介绍，并指导用户如何正确地使用问卷调查系统的各项功能。

13.2.2 问卷编辑页

一份完整的问卷一般由若干个问题项组成。用户可以创建新的问卷，也可以编辑已有问卷。为了方便用户的使用，大多数问卷调查系统都会预设多种问题组件供用户选择。本书的例子精简了问题组件的数量，选取了四类较为常用且有代表性的组件类型，分别为文本题、单选题、多选题和分值题类型。同时系统还提供了问卷大纲功能，用户在创建和编辑问卷的过程中可以随时查看问卷已创建的问题项列表。

13.2.3 我的问卷页

在问卷调查系统中，用户可以查看所创建的问卷列表，以及每份问卷的名称、创建时间、发布时间和当前状态等基本信息，同时还可以查看问卷的统计信息。这里需要特别说明的是，为了简化示例的复杂度，本书精简了问卷统计的功能。除此之外，用户还可以预览、发布及回收问卷等。

13.2.4 用户管理页

问卷调查系统要求用户先注册并登录后才能使用系统相关功能。登录页面是进行用户登录信息验证的入口，用户输入用户名及密码后执行登录操作，验证成功后，就可进入到系统页面。

介绍完问卷调查系统的功能模块之后，接下来将介绍如何设计产品，以及如何将这些特性功能整合到一起。

13.3 产品设计

首页和帮助页都是静态页面，这里不再赘述。本节将主要介绍问卷相关模块的交互设计，如图 13-3 所示。

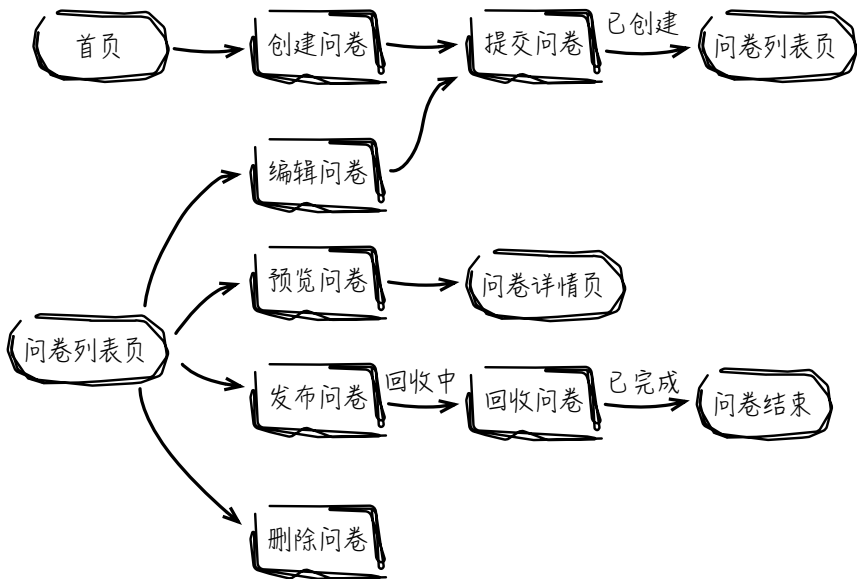


图 13-3 问卷模块交互图

如图 13-3 所示，用户可以创建一个新问卷，也可以编辑一个已有问卷，编辑完成后并提交后跳转到问卷列表页。在问卷列表页，可以预览问卷详情，还可以执行问卷的发布、回收等操作。问卷列表页中的问卷有“已创建”“回收中”和“已完成”三种状态，它们之间的转换关系如下：

- 用户创建或编辑完成后，问卷进入到“已创建”状态。

- 用户发布问卷后，问卷由“已创建”进入到“回收中”状态，此时可以将问卷链接发出去由外部用户填写。
- 问卷调查完成后，用户可以回收问卷，问卷由“回收中”进入到“已完成”状态。

这里需要特别指出的是，对于“已创建”状态下的问卷是可以重新编辑的，而另外两种状态下的问卷则被锁定，不可再次编辑。另外，问卷编辑和问卷列表页面都要求用户是“已登录”状态，否则将跳转到登录页面。

至此，读者应该对将要实现的系统中各个功能模块有了基本了解，关于各个功能模块的细节和实现，将会在后续的项目实施章节中具体阐述。需要再次说明的是，和大多数实际使用中的在线问卷调查系统相比，本书第三部分的示例简化了一些重要功能，在实际使用的问卷调查系统中，它的问题项类型是多种多样的，而示例中只选择性地实现了最具代表性的四种类型，同时也忽略了问卷的分页功能及问卷的回收统计功能，并且产品展示的统计信息均是伪造的。

13.4 小结

本章主要对将要实现的问卷调查系统进行了整体性介绍。首先通过案例背景的介绍，了解了问卷调查系统的现状；然后介绍了我们要实现的问卷调查系统的大体功能和主要特性；最后通过产品交互设计的介绍，进一步了解了问卷调查系统中各个功能模块的交互逻辑关系。

接下来就要学习如何使用 Angular 相关技术来实现问卷调查系统的各个功能特性，让我们继续下一章内容的学习吧。

14

项目起步

上一章介绍了问卷调查系统项目的背景、产品的主要特性及需求分析，相信读者对该系统功能已经有了比较清晰的认识。本章开始将进入问卷调查系统的具体开发，将会讲述项目的技术选型，包括前端、后端、数据库等方面所选用的技术及原因，然后从零开始搭建整个问卷调查系统的开发环境，让读者对于如何开展新项目的研发有更深入的认识。

14.1 Angular CLI

14.1.1 简介

在大中型项目的开发中，一般会选择合适的脚手架快速搭建开发环境。为了给开发者提供更多的便利，Angular 官方推出了自动化命令工具包 Angular CLI。

Angular CLI 可以用来快速创建 Angular 项目，并完成模块安装等初始化工作。同时依托控制台命令可添加模块、组件、管道等模板文件，并且能够执行诸如测试、打包和发布等任务。使用 Angular CLI 创建项目可以避免开发者进行冗繁的构建工作，把主要精力用在业务代码的开发上。根据 CLI 标准模板开发的应用遵循了官方推荐的编程风格，有利于团队成员间的协作开发。

14.1.2 常用命令介绍

Angular CLI 提供了一系列命令，用来提升开发者的工作效率。接下来我们将简单介绍几个常用的命令，以及相关参数的含义。

ng new

new 命令可以用来创建一个新的 Angular 应用。使用方法如下：

```
$ ng new testApp
```

该命令会在 testApp 目录下创建一个名为 test-app（package.json 配置文件中的 name 属性值）的 Angular 应用，同时完成相关的初始化工作。创建后的目录结构如下：

```
testApp
├── README.md
├── e2e
│   ├── app.e2e-spec.ts
│   ├── app.po.ts
│   └── tsconfig.e2e.json
├── karma.conf.js
├── package.json
├── protractor.conf.js
├── .angular-cli.json
├── src
│   ├── app
│   ├── assets
│   ├── environments
│   ├── favicon.ico
│   ├── index.html
│   ├── main.ts
│   ├── polyfills.ts
│   ├── styles.css
│   ├── test.ts
│   ├── tsconfig.app.json
│   ├── tsconfig.spec.json
│   └── typings.d.ts
├── tsconfig.json
└── tslint.json
```

new 命令还支持多个参数的使用，接下来将逐一介绍。

directory

```
$ ng new testApp --directory ngDemo
```

使用 `directory` 参数后，应用的名称依旧是 `test-app`，只不过将所创建的应用放到 `ngDemo` 目录下。



此处的参数也可以使用 `dir` 缩写形式，通常只需要一个减号“-”，即 `ng new testApp -dir ngDemo`，下同。

dry-run

```
$ ng new testApp --dry-run
```

在执行 `new` 命令的时候指定了 `dry-run` 参数，终端会展示该命令将会创建的文件，但不会真正创建。如果需要执行文件的创建，则可以不指定该参数，或者指定参数值为 `false`。`dry-run` 参数可缩写为 `d`。

inline-style

在默认情况下，CLI 创建的组件模板都是使用 `styleUrls` 指定外联样式的，同时创建一个空白的外联样式文件，如 `src/app/app.component.css`。示例代码如下：

```
// src/app/app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

若使用了 `inline-style` 参数，样式文件将不再被创建，而使用内联样式形式，命令如下：

```
$ ng new testApp --inline-style
```

相应的组件代码如下：

```
// src/app/app.component.ts
```

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: []
})
export class AppComponent {
  title = 'app works!';
}
```

在组件元数据定义中，通过 `styles` 可以指定组件使用的内联样式。`inline-style` 参数可缩写为 `is`。

style

在讲解 `inline-style` 时，提到了组件默认使用的样式文件（`src/app/app.component.css`）。此外，还可以通过 `style` 参数修改样式文件使用的后缀名。比如，想要使用 SCSS 格式的样式文件，则可以执行如下命令：

```
$ ng new testApp --style=scss
```

此时生成的样式文件名将变为 `app.component.scss`。该参数支持的值包括：`css`、`scss`、`less`、`sass`、`styl` 等。除在创建应用时改变默认使用的样式文件格式之外，还可以通过修改 `.angular-cli.json` 配置文件中的 `defaults.styleExt` 来实现。



`.angular-cli.json` 是 Angular 应用中非常重要的配置文件，用来存放全局的应用程序配置信息，执行 `ng new` 命令时会自动创建。在接下来介绍前端环境搭建时会详细介绍。

inline-template

和 `inline-style` 参数类似，`inline-template` 指定使用内嵌模板，命令如下：

```
$ ng new testApp --inline-template
```

执行该命令将不再创建模板文件（`src/app/app.component.html`），而是在组件元数据定义中，通过 `template` 属性内嵌 HTML 标签。组件代码如下：

```
// src/app/app.component.ts
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  template: `
    <h1>
      {{title}}
    </h1>
  `,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

inline-template 的缩写为 it。

minimal

指定 minimal 参数可以生成一个最小功能集的 Angular 应用，命令如下：

```
$ ng new testApp --minimal
```

使用了 minimal 参数之后，生成的文件目录如下：

```
testApp
├── package.json
├── src
│   ├── app
│   ├── assets
│   ├── environments
│   ├── index.html
│   ├── main.ts
│   ├── polyfills.ts
│   ├── styles.css
│   ├── tsconfig.app.json
│   └── typings.d.ts
└── tsconfig.json
```

通过和上文讲解 new 命令时生成的目录对比，可以看出使用该参数后，说明文件（README.md）、与测试相关的目录（e2e）和文件（arma.conf.js、tsconfig.spec.json），以及与代码检测相关的文件（tslint.json）等均被移除了。这些目录和文件的共同点就是去掉之后并不会影响应用的运行。

prefix

指定 `prefix` 参数可以生成组件对应的选择器名称的前缀，默认为 `app`。比如执行如下命令：

```
$ ng new testApp --prefix=demo
```

命令执行完成后，打开 `app.component.ts` 文件，代码如下：

```
// src/app/app.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'demo-root', // 由 app-xxx 变为 demo-xxx
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'demo';
}
```

指定了前缀之后，根组件的选择器名称就由原来的 `app-root` 变成了 `demo-root`，后续生成的其他组件的选择器名称前缀也将默认为 `demo`。当然，也可以在配置文件（`.angular-cli.json`）中修改该默认前缀（`app[0].prefix`）。`prefix` 的缩写为 `p`。

routing

在默认情况下，`new` 命令是不会生成路由模块的，如果需要该模块，则可以通过指定 `routing` 参数实现。命令如下：

```
$ ng new testApp --routing
```

此时在创建的 `app` 目录下，将会多出一个 `app-routing.module.ts` 文件。

skip-install 和 skip-tests

顾名思义，指定 `skip-install` 参数在创建项目时将跳过依赖模块的安装，即 `npm install` 的执行过程。其缩写形式为 `si`。而指定 `skip-test` 参数则在创建项目时不再生成与测试相关的 `spec` 文件，以及不再加入 `e2e` 测试功能。其缩写形式为 `st`。命令分别如下：

```
$ ng new testApp --skip-install
$ ng new testApp --skip-tests
```

verbose

指定 `verbose` 参数将在命令执行界面输出命令执行的详情日志，命令如下：

```
$ ng new testApp --verbose
```

`verbose` 的缩写为 `v`。

ng generate

`generate` 命令可以用来创建组件、指令、枚举、模块、服务等文件模板。以组件为例，执行如下命令：

```
$ ng generate component test
```

执行该命令后，会在当前目录下创建一个名为 `test` 的文件夹，并包含了组件（`test.component.ts`）、模板（`test.component.html`）、样式（`test.component.css`）、单元测试（`test.component.spec.ts`）共 4 个文件。

`generate` 命令也支持多个参数的使用，接下来将同样以创建组件为例进行说明。

dry-run

和 `new` 命令的 `dry-run` 参数的功能类似，终端会展示 `generate` 命令将会创建的文件，但不会真正创建。如果需要执行文件的创建，则可以不指定该参数，或者指定参数值为 `false`。`dry-run` 参数的缩写为 `d`。命令如下：

```
$ ng generate component test --dry-run
```

lint-fix

文件生成后，使用 `lint-fix` 修正代码，使代码符合官方推荐的编码风格。命令如下：

```
$ ng generate component test --lint-fix
```

读者也可以通过修改配置文件（`.angular-cli.json`）的配置项（`defaults.lintFix`）来实现每次文件生成后都执行代码修正的工作。`lint-fix` 参数的缩写为 `lf`。

ng build

`build` 命令是 Angular CLI 中使用最多也最重要的命令之一，该命令将应用程序构建到指定的目录下（默认是 `dist`），构建工具是 `webpack`。`build` 命令也可以搭配不同的参数使用，各参数说明如下。

target

执行 build 命令时可以通过参数 target 来指定构建目标，默认的构建目标是开发环境（development），如果想要构建生产环境使用的代码（如压缩混淆后的代码），则可以执行如下命令：

```
$ ng build --target=production
```

environments

每个构建目标都对应着一个环境变量配置文件，用于配置不同的环境变量。这些配置文件统一存放在根目录下的 environments 文件夹中，如生产环境对应的环境变量配置文件为 environment.prod.ts。构建目标和环境变量配置文件之间的对应关系在 angular-cli.json 文件的 environments 属性中。示例代码如下：

```
"environmentSource": "environments/environment.ts",  
"environments": {  
  "development": "environments/environment.ts",  
  "production": "environments/environment.prod.ts"  
}
```

开发者也可以使用简写方式，如使用 dev 来代替 development，用 prod 代替 production。

```
"environments": {  
  "dev": "environments/environment.ts",  
  "prod": "environments/environment.prod.ts"  
}
```

除此之外，我们还可以添加自定义的环境变量配置文件和构建目标。例如项目中需要一个准生产环境（使用 uat 表示），则可以创建一个名为 environment.uat.ts 的准生产环境变量文件，并在 angular-cli.json 文件中配置关联关系。示例代码如下：

```
"environmentSource": "environments/environment.ts",  
"environments": {  
  "dev": "environments/environment.ts",  
  "uat": "environments/environment.uat.ts",  
  "prod": "environments/environment.prod.ts"  
}
```

接下来就可以使用 uat 作为构建目标生成准生产环境下的代码文件了。以下三个命令是等价的：

```
ng build --target=production --enviroment=prod
ng build --prod --env=prod
ng build --prod
```

build 命令除支持 target 和 environments 参数之外，还支持其他参数，比如：

- base-href 参数可以指定构建后的 <base> 标签，用来指定页面上所有链接使用的默认地址路径。
- aot 参数指定以 AoT 的方式构建应用程序。关于 AoT 的构建方式将在下文中详细介绍。
- output-path 参数指定构建后的文件输出路径。
- watch 参数指定是否监听文件变化并重新构建。

ng serve

serve 命令首先会执行 build 命令来完成应用程序的构建，然后再启动一个 Web 服务。因此 build 命令支持的参数都可以用在 serve 命令中。例如，在生产环境中启动服务的命令如下：

```
$ ng serve --prod
```

在 Angular CLI 1.5.0（TypeScript 2.3）之后，ng serve 已经支持 aot 参数，极大地提升了开发环境的构建性能，并且在后续的版本中 aot 参数可能变为默认参数。

除此之外，serve 命令还支持如下参数：

- live-reload 参数可以标识当代码更新后，页面是否需要自动刷新。
- ssl 参数指定是否支持 https。
- open 参数标识服务启动成功后，是否需要在默认浏览器中打开。

ng get/set

顾名思义，get 和 set 命令分别用来获取和设置配置信息，这里的配置信息一般存放在 .angular-cli.json 中。若想获取当前应用程序的基本信息，就可以执行如下命令：

```
$ ng get project
```

该命令会返回应用程序的名称、版本号、作者（如果有配置的话）等信息。同样地，也可以使用 set 命令修改配置信息，例如执行如下命令修改应用程序的名称：

```
$ ng set project.name=newName
```

`get` 和 `set` 命令还支持 `global` 参数, 该参数用来获取和设置 Angular CLI 的全局配置信息, 该配置信息一般存放在当前系统用户的 `home` 目录下。例如, 想要忽略执行 Angular CLI 命令时的警告信息, 就可以执行如下命令:

```
$ ng set --global warnings.nodeDeprecation=false
```

ng eject

`eject` 命令将应用程序内置的 Webpack 配置文件 (`webpack.config.js`) 和 `npm` 执行脚本 (参见 `package.json` 文件中的 `script` 属性) 释放出来, 并存储在根目录下, 以方便开发者自定义配置。如执行以下命令:

```
$ ng eject
```

`eject` 命令执行完成后, 终端会显示如下输出:

```
=====
Ejection was successful.
```

To run your builds, you now need to do the following commands:

- "npm run build" to build.
- "npm run test" to run unit tests.
- "npm start" to serve the app using webpack-dev-server.
- "npm run e2e" to run protractor.

Running the equivalent CLI commands will result in an error.

```
=====
Some packages were added. Please run "npm install".
```

通过输出结果不难看出, 部分 `ng` 命令将不能再使用, 例如需要使用 `npm run build` 命令替代 `ng build` 来完成应用程序的构建。另外, `eject` 命令执行后将引入新的依赖包, 还需要执行 `npm install` 命令来安装这些依赖包。

`eject` 命令支持 `build` 命令的所有参数, 执行带有相关参数的 `eject` 命令后, 输出的配置文件等价于执行了相关 `build` 命令之后的应用程序配置。例如执行如下命令, 将输出生产环境下的 Webpack 配置:

```
$ ng eject --prod
```

`eject` 还支持一个特有的 `force` 参数, 该参数将强制覆盖项目里已有的 Webpack 配置文件和 `npm` 命令脚本。

需要特别说明的是，`eject` 命令执行后是不可逆的，Angular CLI 并没有提供还原命令，一旦执行将无法再恢复到应用程序最初的状态，所以开发者需要谨慎使用。虽然 Angular CLI 没有提供还原方法，但是开发者依然可以手动修改 `angular-cli.json` 文件的 `project.ejected = false` 来达到还原效果。

至此，本章就介绍完了 Angular CLI 最常用的几个命令。除此之外，还有诸如 `lint`、`test`、`e2e` 等与代码检测及测试相关的命令，将在第 18 章中介绍。

14.2 其他技术选型

14.2.1 UI 样式库

确定好脚手架之后，接下来选择一款合适的 UI 样式库，用来帮助开发者更便捷地开发应用。适合 Angular 的 UI 样式库，业界提供了多种可选择方案，其中以 Angular 官方的 Material 尤为抢眼，但截止到本书写作时还不太成熟，一些组件与文档并不齐全，而且问卷调查系统本身页面的 UI 复杂度要求不高，因此本书不使用 Material，而是使用 `ngx-bootstrap`。

Bootstrap UI 样式库是 Twitter 推出的一款前端 UI 框架，使用 Bootstrap 可以快速地开发出好看的 UI 界面。`ngx-bootstrap` 是对 Bootstrap 进行 Angular 包装的组件和指令集合，需要配合 Bootstrap 3 或者 Bootstrap 4 使用。它为开发者提供了丰富多样的 UI 组件和界面样式，例如在本书的问卷调查系统中使用的按钮、轮播图和警告框等样式。

14.2.2 后端服务器

介绍完前端技术的选型后，在这个系统中，还需要使用后端服务来存储数据或者实现登录鉴权等功能。本书主要讲解 Angular 这个前端框架，为了避免后端场景的复杂化，尽量使用简化的后端开发方式，在本书的第三部分将选择 `json-server` 这个简易的 Node 模块来模拟后端服务。在真实的应用场景中，通常会使用 Node.js + MongoDB 或者 PHP + MySQL 等组合来提供后端服务，提供 API 与前端进行通信及数据持久化。`json-server` 是用来模拟服务器提供 API 服务的，它没有实际的数据库，而是采用纯 JSON 文本的方式存储数据，通过 `json-server` 能以少量的代码搭建一个 REST API 服务。关于 `json-server` 的具体用法，读者可以参考其官方 GitHub 上的文档，后面也会详细介绍在本项目中如何使用，这里就不展开介绍了。

以上就是整个问卷调查系统的技术选型。首先采用 Angular CLI 作为前端 Angular 的脚手架，快速完成项目的搭建；接着采用 `ngx-bootstrap` 作为前端 UI 样式库，用来帮

助我们快速实现页面的原型效果；最后采用 json-server 作为后端服务器，它简单易用，让我们更专注于 Angular 项目的开发。

14.3 环境搭建

了解了问卷调查系统的技术选型与相关技术栈之后，接下来将介绍如何整合这些技术来快速搭建开发环境。

14.3.1 搭建前端环境

在搭建开发环境之前，首先需要全局安装 Angular CLI。在安装之前，需要确认 Node 和 npm 版本分别为 6.x.x 和 3.x.x 或以上。安装命令如下：

```
$ npm install -g @angular/cli
```

接下来创建项目的主目录 angular-questionnaire，进入该目录下，执行如下命令创建新项目：

```
$ ng new frontend
```

该命令会创建运行 Angular 应用所需的基本文件，并安装所有依赖的 npm 包。安装完成后，进入 frontend 目录下，启动服务：

```
$ ng serve
```

此时在浏览器中打开地址 <http://localhost:4200/>，如果看到如图 14-1 所示的界面，就代表服务启动成功。

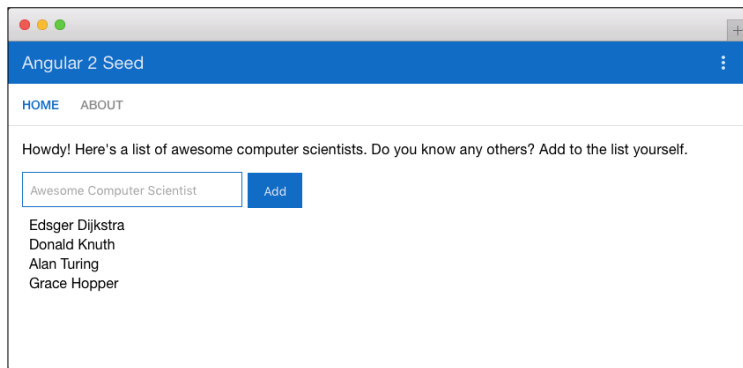


图 14-1 CLI 初始界面

frontend 的目录结构如下：

```

.
├── README.md
├── e2e                <- 端到端测试文件的存放目录
├── karma.conf.js     <- 测试启动的配置文件
├── node_modules      <- 应用程序依赖包的存放目录
├── package.json      <- 项目的依赖配置
├── protractor.conf.js <- 端到端测试的配置文件
├── src               <- 程序主目录
├── tsconfig.json     <- TypeScript 相关配置
└── tslint.json       <- 代码质量检测配置

```

14.3.2 引入样式库

项目搭建好之后，接下来引入 ngx-bootstrap 样式库。

首先安装 ngx-bootstrap，因为 ngx-bootstrap 依赖 Bootstrap 样式库，所以需要一并安装。在 frontend 目录下执行如下命令：

```
$ npm install ngx-bootstrap bootstrap --save
```

安装成功后，接下来需要将 Bootstrap 的样式文件引入到工程中。打开文件 angular-cli.json，将样式文件添加到 styles 配置项中，代码如下：

```

"styles": [
  "../node_modules/bootstrap/dist/css/bootstrap.min.css",
  "styles.css"
]

```

然后就可以在项目中使用 ngx-bootstrap 提供的组件了。以添加一个提示框为例，首先导入提示框模块。示例代码如下：

```

import { AlertModule } from 'ngx-bootstrap';
...

@NgModule({
  //...
  imports: [AlertModule.forRoot(), ... ],
  //...
})

```

接下来就可以在组件模板中以标签的形式使用提示框了。示例代码如下：

```
<alert type="success">hello</alert>
```



type 是 Alert 组件的属性。

14.3.3 搭建后端环境

在上述两个章节的内容中，我们通过 Angular CLI 和 ngx-bootstrap 完成了前端开发环境的搭建，接下来将介绍如何基于 json-server 快速搭建一个简单的后端开发环境。

首先在根目录 angular-questionnaire 下创建 backend 文件夹，用来存放后端代码。接下来进入 backend 文件夹，初始化后端开发环境，生成 package.json 文件，执行以下命令：

```
$ npm init
```

然后根据命令行的提示来配置相应的参数，也可以一直按回车键生成默认配置。接下来安装 json-server、body-parser 和 lowdb 依赖库并将配置保存到 package.json 文件中。安装命令如下：

```
$ npm install --save json-server body-parser lowdb@0.12.2
```

body-parser 是 Node 服务器中常用的中间件，主要用来提前解析请求体的内容，然后就可以在业务代码中通过 req.body 的方式读取请求体内容了。lowdb 是一个基于 Node 的 JSON 文件数据库，在本项目中用于存储问卷调查系统数据。最后在配置中将 main 选项所代表的入口文件修改为 app.js，配置如下：

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.14.2",
```

```
"json-server": "^0.8.7",  
"lowdb": "^0.12.2"  
}  
}
```

最后创建入口文件 `app.js`，并添加初始化代码。示例代码如下：

```
const jsonServer = require('json-server');  
const bodyParser = require('body-parser');  
const low = require('lowdb');  
const storage = require('lowdb/file-async');  
  
// 创建一个 Express 服务器  
const server = jsonServer.create();  
  
// 使用 json-server 默认选择的中间件（logger、static、cors 和 no-cache）  
server.use(jsonServer.defaults());  
// 使用 body-parser 中间件  
server.use(bodyParser.json());  
  
// 数据文件  
const dbfile = process.env.prod === '1' ? 'db.json' : '_db.json';  
// 创建一个 lowdb 实例  
const db = low(dbfile, {storage});  
  
// 路由配置  
const router = jsonServer.router(dbfile);  
server.use('/api', router);  
  
// 启动服务，并监听 5000 端口  
server.listen(5000, () => {  
  console.log('server is running at ', 5000, dbfile);  
});
```

接着执行启动命令：

```
$ node app.js
```

启动命令执行成功后，就可以在命令行界面看到如下信息：

```
server is running at 5000 _db.json
```

后端开发环境主要是为了辅助完成整个项目的搭建和启动，不是本书需要重点介绍的内容，所以这里就不展开介绍了，感兴趣的读者可以参考 json-server 的官方文档。另外，关于前端对后端各逻辑接口的调用，将在后续实现细节时介绍。

14.4 目录结构介绍

搭建好前后端开发环境后，接下来就正式进入项目各模块功能的开发了。在此之前，先看一下目前的工程目录结构：

```
.
├── backend
├── frontend
│   ├── README.md
│   ├── e2e
│   ├── karma.conf.js
│   ├── npm-debug.log
│   ├── package.json
│   ├── protractor.conf.js
│   ├── src
│   │   ├── app
│   │   ├── assets
│   │   ├── environments
│   │   ├── favicon.ico
│   │   ├── index.html
│   │   ├── main.ts
│   │   ├── polyfills.ts
│   │   ├── styles.css
│   │   ├── test.ts
│   │   ├── tsconfig.app.json
│   │   ├── tsconfig.spec.json
│   │   └── typings.d.ts
│   ├── tsconfig.json
│   └── tslint.json
```

在上述目录结构中，frontend 为前端目录，backend 为后端目录。在 frontend 目录中，与项目相关的业务代码都存放在 src 文件夹中，主要目录和功能说明如下。

- app 目录：用来存放项目的业务逻辑代码文件。
- index.html：首页入口文件。

- main.ts: 应用程序入口文件。
- assets 目录: 用来存放图片、SVG 文件等静态资源。
- environments 目录: 用来存放不同环境下的变量配置文件。
- styles.css: 全局样式文件。
- polyfills.ts: 兼容性适配文件, 主要有两个用处, 一是兼容低级浏览器; 二是引入 Zone.js。

下面简单介绍 main.ts 入口文件, 初始代码如下:

```
// main.ts
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

// 开启生产模式, 会关闭一些额外的检查工作, 提升运行时性能
if (environment.production) {
  enableProdMode();
}

// 编译并启动根模块 AppModule
platformBrowserDynamic().bootstrapModule(AppModule);
```

了解了几个重要的目录和文件后, 接下来开始进入正式的项目开发, 首先从简单的首页开始。

14.5 首页开发

首先创建 home 组件。进入 frontend 目录下, 执行如下命令:

```
$ ng generate component home
```

该命令将在 app 目录下新增一个 home 目录, 并创建与组件相关的文件如下:

```
.
├── home.component.css  <- 样式文件
├── home.component.html <- 模板文件
├── home.component.spec.ts <- 单元测试文件
└── home.component.ts   <- 组件文件
```

同时, home 组件会被自动导入到 app.module.ts 中。我们首先要实现的是 HomeComponent 首页组件。示例代码如下:

```
// home/home.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'sd-home',
  styleUrls: [ 'home.component.css' ],
  templateUrl: 'home.component.html'
})
export class HomeComponent {

  slides: Array<any> = [];
  slogan: Array<String> = [
    '免费简约的问卷系统',
    '简单 好用 在线调查网站',
    '多方式创建编辑问卷'
  ];

  constructor() {
    for (let i = 0; i < 3; i++) {
      this.addSlide(i);
    }
  }

  addSlide(idx: number) {
    this.slides.push({
      image: `./assets/img/home/banner_${idx+1}.jpg`,
      text: this.slogan[idx]
    });
  }
}
```

HomeComponent 组件类定义了一个 slides 数组, 该数组存放的是轮播图片的数据, 并提供了 addSlide() 方法来填充 slides 数组数据。

轮播图片是使用 Bootstrap 提供的 Carousel 和 Slide 组件实现的, Home 模板示例代码如下:


```
<!-- home/home.component.html -->

<div class="home">
  <carousel>
    <slide *ngFor="let slidez of slides; let index=index"
      [active]="slidez.active">
      <div class="slide-image">
        <img [src]="slidez.image">
      </div>

      <div class="carousel-caption">
        <p>{{slidez.text}}</p>
      </div>
    </slide>
  </carousel>

  <section>
    <h2 class="hero-title ">
      <span>10,145,267</span>个用户正在使用
    </h2>
  </section>
  <div class="divider"></div>
</div>
```

为了使用该组件，首先需要在模块中导入它，所以还需要创建首页的模块文件，命令如下：

```
$ ng generate module home
```

命令执行完成之后，就可以在 home 目录下看到首页的模块文件了。接下来需要在 HomeModule 模块中导入轮播组件所属模块（CarouselModule）。示例代码如下：

```
// home/home.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { HomeComponent } from './home.component';
import { CarouselModule } from 'ngx-bootstrap/components/carousel';
import { SharedModule } from '../shared/shared.module';
```

```
@NgModule({
  imports: [CommonModule, SharedModule, CarouselModule],
  declarations: [HomeComponent],
  exports: [HomeComponent]
})
export class HomeModule { }
```

在模块文件中，imports 用于导入依赖的其他模块，这里导入了 CommonModule、SharedModule 和 CarouselModule 模块，其中 CommonModule 包含了 Angular 常用的基本指令，CarouselModule 是 ngx-bootstrap 提供的与轮播图片组件相关的模块。declarations 指明了 HomeModule 模块实现的组件、指令、管道等，这里设置为刚创建的 HomeComponent 组件；exports 定义了模块暴露出去的部分，如此处的 HomeComponent，以便其他模块能使用 HomeComponent 组件。

我们刷新浏览器就可以看到首页更新后的界面，增加了图片轮播展示功能，效果如图 14-2 所示。



图 14-2 首页轮播图片

接下来用同样的思路和方法，开发 About 帮助页组件和模块，它使用 ngx-bootstrap 提供的 AccordionComponent 组件来实现文本信息的展示和折叠功能。界面效果如图 14-3 所示。



图 14-3 帮助页效果图

至此，我们通过引入 ngx-bootstrap 组件就完成了首页和帮助页的开发。

14.6 导航栏开发

在应用中除会用到类似于首页和帮助页等功能特性组件之外，还会经常用到一些诸如导航栏（Navbar）和工具栏（Toolbar）等公用组件，推荐将这些公用组件放到核心模块里。创建核心模块的命令如下：

```
$ ng generate module core
```

命令执行完成后，会生成 core 文件夹及 core 模块文件，接下来在 core 目录下添加导航栏组件，命令如下：

```
$ ng generate component core/navbar
```

添加导航栏组件的代码如下：

```
// core/navbar.component.ts
```

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'sd-navbar',
  templateUrl: 'navbar.component.html',
  styleUrls: ['navbar.component.css'],
})
export class NavbarComponent {
  constructor(private router:Router) { }
}
```

NavbarComponent 依赖 Router 服务，在构造函数中引入 Router 实例。接下来修改模板文件代码：

```
<nav class="nav navbar-nav">
  <a [routerLink]="['/']" [routerLinkActive]='["router-link-active"]' [
    routerLinkActiveOptions]={exact:true}">首页</a>
  <a [routerLink]="['/about']" [routerLinkActive]='["router-link-active"]' [
    routerLinkActiveOptions]={exact:true}">帮助</a>
</nav>
```

这里使用了 Router 模块中定义的 routerLink、routerLinkActive、routerLinkActiveOptions 等多个属性，感兴趣的读者可以参考第 11 章路由章节来了解更多内容，这里不再赘述。

还需要注意的是，在特性模块目录下，可能会出现 shared 目录，用来存放那些只在当前特性模块中才需要用到的组件与服务等。同时，部分模块可以封装出自己的子路由模块，来实现比较复杂的路由管理及页面展示控制，这些也可以存放在特性模块目录下，在第 16 章中将进行更加具体的介绍。

14.7 小结

本章主要介绍了项目起步的相关事宜。首先介绍了项目的技术选型，包括 Angular 提供的命令行工具 Angular CLI、UI 样式库 ngx-bootstrap 和后端服务器 json-server；然后介绍了环境的搭建，以及如何使用这些技术；最后通过一个简单的首页和导航栏的例子讲解了模块、路由、组件、模板的创建和使用。下一章将讲解问卷调查系统编辑页的实现。

15

问卷编辑模块

上一章介绍了与项目起步相关的内容，接下来将正式进入项目开发阶段。问卷调查系统由很多功能模块组成，其中问卷编辑模块的功能最复杂，它涉及多个组件，是整个系统的核心模块。

本章将着重介绍问卷编辑模块的相关内容，包括模块、组件的组成和数据模型的定义等。

15.1 概述

正如在第 6 章组件章节中所提到的，Angular 采用的是基于组件模块化的开发方式，组件或者模块之间并不是孤立存在的。组件之间存在着支持数据流动的层级关系，这种关系最终以组件树的形式表现出来。而组件又可以形成一个新的模块，因此关于组件（模块）树的设计是整个大模块架构设计非常关键的一环。

15.1.1 特性管理模块

相对于首页及帮助页等一般的特性模块而言，问卷编辑和我的问卷模块拥有更多的管理特性，因此需要控制访问权限。在讲解问卷编辑功能模块之前，我们先引入一个名为 AdminModule 的特性管理模块，问卷编辑模块将置于该模块之下统一管理。目录结构如下：

app/admin

```

|—— admin-routing.module.ts
|—— admin.component.html
|—— admin.component.ts
└—— admin.module.ts

```

AdminModule 模块用来管理 admin 目录下定义的各类组件，其中 AdminComponent 是 AdminModule 的入口组件，AdminRoutingModule 用来管理 admin 下所有子页面的路由配置。接下来，在该模块里添加用户编辑组件（edit.component.ts），完整的目录结构如下：

app/admin

```

|—— admin-routing.module.ts
|—— admin.component.html
|—— admin.component.ts
|—— admin.module.ts
└—— edit
    |—— edit.component.css
    |—— edit.component.html
    |—— edit.component.ts
    |—— index.ts
    └—— shared

```

EditComponent 组件类目前是一个空类，在下面的章节中将逐渐完善它。细心的读者可能已经发现，这里并没有具体实现编辑页及对应的路由模块，只是在 AdminModule 和 AdminRoutingModule 中集中管理 admin 目录下的组件和路由。其中 AdminModule 的代码如下：

```
// admin/admin.module.ts
```

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { AdminComponent } from './admin.component';
import { EditComponent } from './edit';

import { AdminRoutingModule } from './admin-routing.module';

@NgModule({

```

```
imports: [CommonModule, AdminRoutingModule],
declarations: [AdminComponent, EditComponent]
})
export class AdminModule { }
```

如上述代码所示，编辑页面组件（EditComponent）和 Admin 路由模块（AdminRoutingModule）均通过 Admin 模块（AdminModule）引入和声明。

AdminRoutingModule 路由模块的代码如下：

```
// admin/admin-routing.module.ts

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { AdminComponent } from '../admin.component';
import { EditComponent } from '../edit/edit.component';

const routes: Routes = [{
  path: 'admin',
  component: AdminComponent,
  children: [{
    path: '',
    children: [{
      path: 'edit/:id',
      component: EditComponent
    }]
  }]
}];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AdminRoutingModule { }
```

在路由模块中，首先定义了一个路由数组，并将编辑页面的路径定义在子路由数组中，然后使用路由模块的 forChild() 方法使其生效。

最后介绍一下 AdminComponent 的模板，子路由指令（router-outlet）定义在该模板里。示例代码如下：

```
<!-- admin.component.html -->

<sd-navbar></sd-navbar>
<router-outlet></router-outlet>
```

在 AdminComponent 组件模板中，首先添加导航栏标签，这样所有 /admin 路由下的页面顶部都将展示出导航栏；其次，添加 <router-outlet> 指令标签，所有子路由中的组件都将被渲染到该标签处。



关于 forChild() 方法和 <router-outlet> 标签的更多使用介绍，请参考本书第 11 章“路由”部分。

15.1.2 功能设计

在问卷编辑页中可以创建新的问卷，也可以编辑已有的问卷。在问卷编辑页上可以添加新的问题项，且为了能及时查看已添加的问题项，问卷编辑页还提供了问卷大纲功能。在问卷的核心部分可以编辑问卷的开始语、结束语及所包含的问题项。问卷编辑页组件为 EditComponent，它包含了问题选择组件（QuestionSelectComponent）、问卷大纲组件（QuestionnaireOutlineComponent）和问卷组件（QuestionnaireComponent）三个子组件，其中问卷子组件下包含有不同类型的问题组件，如单选问题组件（QuestionRadioComponent）等。问卷编辑模块的组件设计如图 15-1 所示。

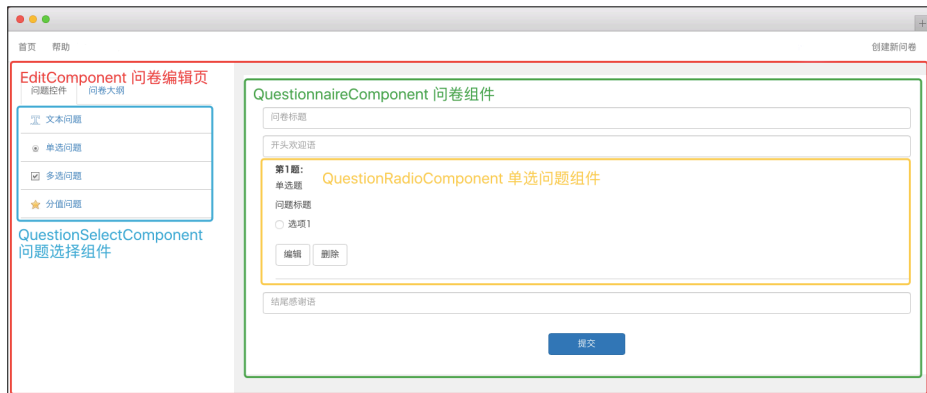


图 15-1 问卷编辑页

问卷大纲组件（QuestionnaireOutlineComponent）在图中并没有列出来，点击问卷大纲标签即可显示出来。问卷编辑页面的完整目录结构如下：


```

app/edit
├── edit.component.ts
├── edit.component.html
├── edit.component.css
├── edit.module.ts
├── edit.routes.ts
├── index.ts
├── shared
│   ├── edit-shared.module.ts
│   ├── index.ts
│   ├── question-select
│   │   ├── question-select.component.ts
│   │   ├── question-select.component.html
│   │   ├── question-select.component.css
│   │   └── index.ts
│   └── questionnaire-outline
│       ├── questionnaire-outline.component.ts
│       ├── questionnaire-outline.component.html
│       ├── questionnaire-outline.component.css
│       └── index.ts

```

至此，相信读者对问卷编辑功能涉及的组件，以及它们之间的关系有了大概的了解。接下来将介绍在组件开发中会使用到的数据模型，以帮助读者更好地了解组件间数据的流动和前后端之间的数据交互。

15.1.3 数据模型

编辑页主要涉及两种核心的数据模型，即问卷模型和问卷问题模型。这两种数据模型的定义将在多个特性模块中被使用到，因此我们将相关的文件放到 `frontend/src/app/shared/models` 共享目录下。

问卷模型

问卷模型描述了一个调查问卷的基本数据结构，其完整的数据结构定义如下：

```

// shared/model/questionnaire.models.ts

import { QuestionModel } from './question.model';

export class QuestionnaireModel {

```

```
id?:string;
title:string;
starter:string;
ending:string;
state:QuestionnaireState;
questionList: QuestionModel[];
createDate?:string;
}
```

问卷数据模型定义的属性说明如表 15-1 所示。

表 15-1 问卷数据模型的属性说明

属性名称	含义	类型
id	问卷的唯一性标识	字符串
state	标识问卷当前所处的状态	枚举
title	问卷的标题	字符串
starter	问卷开头语	字符串
ending	问卷结尾语	字符串
createDate	问卷创建日期	字符串
questionList	问卷包含的问题列表	数组

其中问卷标识 id 和创建日期 createDate 都是非必选属性，这是因为问卷标识和创建日期是在保存问卷时由后台自动生成的，创建时不需要传入。另外需要特殊说明的是，问卷状态对应的类型 QuestionnaireState 是一个枚举类型，其中的三个枚举项分别对应着问卷的三种状态。

- 已创建状态：表示问卷已经对外发布。
- 回收中状态：表示处于回收问卷填写信息阶段。
- 已结束状态：表示问卷已经结束回收，调查通道也已关闭。

问卷状态类型的枚举定义如下：

```
// shared/model/questionnaire.model.ts

export const enum QuestionnaireState {
  Created,
  Published,
```

```
    Finished
  }
```

问卷问题模型

问卷问题是构建一个问卷的基础，问卷问题模型详细地描述了一个完整的问题所需的基本要素，它的数据结构定义如下：

```
// shared/model/question.model.ts

export class QuestionModel {
  title:string;
  type:QuestionType;
  options?:any[];
  answer:any;
}
```

问题数据模型定义的属性说明如表 15-2 所示。

表 15-2 问题数据模型的属性说明

属性名称	含义	类型
title	问题的标题	字符串
type	标识问题的类型	枚举
options	存储答案的选项	数组
answer	存储问题的答案	任意

其中 options 属性为可选的，这是因为不是所有的问题都具备可选择的答案选项。在本书的例子中，单选题和多选题都具有可选的答案选项，而文本题和分值题则无此属性。此外，为了方便扩展，这里并没有为 options 属性设置特定的数据类型，而是使用了 any 关键字标识。

正如在第 13 章中所提到的，本书的问卷调查系统只实现了四种类型的问题，分别为文本题、单选题、多选题和分值题。问题类型的枚举定义如下：

```
// shared/model/question.model.ts

export const enum QuestionType {
  Text,
```

```
    SingleSelect,  
    MultiSelect,  
    Score  
}
```

15.2 问卷编辑模块开发

问卷编辑页面涉及的各文件的初始代码与在第 14 章最后提到的 Home 模块类似，这里就不再详细介绍了。在问卷编辑页面，我们既可以创建一个新问卷，也可以编辑一个现有问卷。无论哪种情况，首先要做的都是选择并添加各种类型的问题项，因此最先需要实现的就是问题选择组件。

15.2.1 问题选择组件

组件开发

由于问题选择组件（question-select.component.ts）只在问卷编辑页面中使用到，所以与该组件相关的文件统一放到 edit/shared/question-select 目录下。组件类的示例代码如下：

```
// edit/shared/question-select/question-select.component.ts  
  
import { Component, EventEmitter, Output } from '@angular/core';  
  
import { QuestionType } from '../../shared/models/question.model';  
  
@Component({  
  selector: 'question-select',  
  styleUrls: ['question-select.component.css'],  
  templateUrl: 'question-select.component.html'  
})  
export class QuestionSelectComponent {  
  
  @Output() addQuestionRequest = new EventEmitter();  
  
  private controls:any[];  
  
  constructor() {  
    this.controls = [  

```

```

    {type: QuestionType.Text, label: '文本问题', iconClass: 'icon-text'},
    {type: QuestionType.SingleSelect, label: '单选问题', iconClass: 'icon-radio'},
    {type: QuestionType.MultiSelect, label: '多选问题', iconClass: 'icon-checkbox'},
    {type: QuestionType.Score, label: '分值问题', iconClass: 'icon-star'}
  ];
}

onAddQuestion(control: any) {
  this.addQuestionRequest.emit(control.type);
}
}

```

组件模板代码如下：

```

<!-- edit/shared/question-select/question-select.component.html -->

<ul class="question-controls list-group">
  <li *ngFor="let control of controls" class="list-group-item">
    <a [ngClass]="control.iconClass" (click)="onAddQuestion(control)">{{control.
      label}}</a>
  </li>
</ul>

```

如上述代码所示，问题选择组件的基本创建步骤如下：

- 在组件中定义一个私有变量 `controls`。该变量是一个数组，保存了四种问题类型，并在构造函数中进行赋值。
- 在对应的组件模板中循环遍历 `controls` 数组，并以列表的形式展示出来。
- 自定义点击绑定事件 `onAddQuestion`，点击链接会触发该事件，并触发输出属性事件 `addQuestionRequest`。

经过前两个步骤，问题选择组件已经可以展示出来了。细心的读者应该可以注意到，在该选择组件的问题列表中每个具体问题类型可触发一个添加问题的点击事件。接下来需要实现的就是将该点击事件传递给问卷编辑组件（父组件），并在问卷编辑页面中添加相应类型的问题，整个过程是通过自定义事件实现的。

接下来就是通过入口文件将组件暴露出去，示例代码如下：

```
// edit/shared/question-select/index.ts
```

```
export * from './question-select.component';
```

Angular 通过 `@NgModule` 中的 `imports` 和 `exports` 元数据管理模块间的引用关系，这里将在 `shared` 目录下创建一个模块文件来统一管理问卷编辑页面中所有的子组件。我们可以将问题选择组件添加到该模块文件中，示例代码如下：

```
// edit/shared/edit-shared.module.ts
```

```
import { NgModule } from '@angular/core';
```

```
import { CommonModule } from '@angular/common';
```

```
import { QuestionSelectComponent } from './question-select/index';
```

```
@NgModule({  
  imports: [CommonModule],  
  declarations: [QuestionSelectComponent],  
  exports: [QuestionSelectComponent]  
})
```

```
export class EditSharedModule { }
```

同时，我们还将在 `shared` 目录下创建一个新的入口文件，用来统一导出所有的子组件（目前只有一个问题选择组件）。示例代码如下：

```
// edit/shared/index.ts
```

```
export * from './question-select/index';
```

添加问题选择组件到问卷编辑页面中

至此，问题选择组件已经开发完毕，下一步就是将它添加到问卷编辑页面中。与问题编辑组件的引入方式类似，组件是在 `Admin` 模块文件中统一引入的。示例代码如下：

```
// admin.module.ts
```

```
import { NgModule } from '@angular/core';
```

```
import { CommonModule } from '@angular/common';
```

```
import { AdminComponent } from './admin.component';
```

```
import { EditSharedModule } from './edit/shared/edit-shared.module';
```

```
import { EditComponent } from './edit';

import { AdminRoutingModule } from './admin-routing.module';

@NgModule({
  imports: [CommonModule, AdminRoutingModule, EditSharedModule],
  declarations: [AdminComponent, EditComponent]
})
export class AdminModule { }
```

当组件引入成功后，接着就可以在问卷编辑模板文件中使用它了。示例代码如下：

```
<!-- edit/edit.component.html -->

<!-- ... -->
<tab heading="问题控件">
  <question-select (addQuestionRequest)="onAddQuestion($event)"></question-select>
</tab>
<!-- ... -->
```

`addQuestionRequest` 事件是由问题选择组件（子组件）传递过来的，我们需要在问卷编辑组件（父组件）中捕获并处理。添加问题的最终结果是将问题添加到问卷的问题列表中，而问卷属性将在问卷组件的开发中介绍到，因此这里将 `onAddQuestion()` 方法的具体实现放到了问卷组件开发之后。示例代码如下：

```
// edit/edit.component.ts

// ...
export class EditComponent {
  constructor(){ }

  onAddQuestion(type: QuestionType){
    // TODO: 添加问题到问卷的问题列表中
  }
}
```

添加问题选择组件的基本步骤总结如下：

- 在模块文件中引入组件所在模块（`EditSharedModule`）。
- 在问卷编辑组件的模板中添加问题选择组件元素标签 `<question-select>`。
- 在问卷编辑组件中捕获并实现子组件传递过来的添加问题的事件（`addQuestionRequest`）。

问题选择组件添加完成之后，问卷编辑页面如图 15-2 所示。

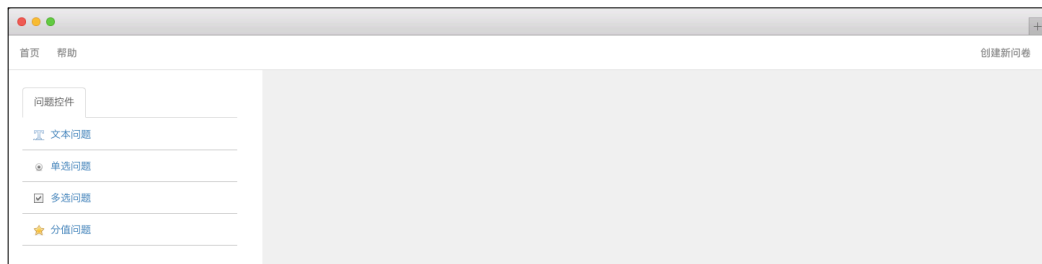


图 15-2 问卷编辑页面



关于父子组件之间的数据传递方法，可以参考第 6 章中的“组件交互”部分。

15.2.2 问题组件

问题组件是问卷组件最重要的组成部分。几乎所有类型的问题组件都会包括问题标题、问题答案和控件状态等属性，以及编辑、取消编辑、保存和删除等操作，因此本例中将这些公共功能封装到一个父类中，具体类型的问题组件都将继承该父类。问题组件在多个页面中都会使用到，所以这里将该父类及相关类型的问题组件放到了 `shared/question` 目录中。示例代码如下：

```
// shared/question/question.component.ts

import { OnInit, EventEmitter } from '@angular/core';

import { QuestionModel } from '../shared/models/question.model';

export class QuestionComponent implements OnInit {
  question: QuestionModel;
  backupQuestion: QuestionModel;
  editable: boolean = false;
  isEditing: boolean = false;
  deleteQuestionRequest: EventEmitter<any> = new EventEmitter();

  ngOnInit() {
    this.copyQuestion();
  }
}
```



```
}

private copy(source: QuestionModel): QuestionModel {
    return <QuestionModel>JSON.parse(JSON.stringify(source));
}

public copyQuestion() {
    this.backupQuestion = this.copy(this.question);
}

onEdit() {
    this.isEditing = true;
}

onSave() {
    this.copyQuestion();
    this.isEditing = false;
}

onCancel() {
    this.question = this.copy(this.backupQuestion);
    this.isEditing = false;
}

onDelete() {
    this.deleteQuestionRequest.emit(this.question);
}
}
```

在问题组件父类中定义了如下几个属性。

- question: 问题数据, 由问卷父组件传入。
- backupQuestion: 问题数据备份, 用于取消编辑时恢复初始数据。
- editable: 标记问题是否可编辑, 由问卷父组件传入。
- isEditing: 标记当前问题是否处于编辑状态。
- deleteQuestionRequest: 删除问题的自定义事件, 该事件最终将传递给问卷父组件。

除问题组件类属性之外，还定义了几个组件类方法，这些方法实现了编辑、保存、取消编辑及初始化等功能，具体说明如下。

- `onEdit()`：单击“编辑”按钮时，进入编辑状态。
- `onSave()`：单击“保存”按钮时，保存并退出编辑状态。
- `onCancel()`：单击“取消”按钮时，恢复问题编辑前的数据，同时退出编辑状态。
- `onDelete()`：单击“删除”按钮时，删除当前问题，该操作通过自定义事件传递到问卷父组件中执行。

除这些方法之外，还定义了一个私有方法，用来做对象的深拷贝。读者可能对有些方法不太理解，在接下来的文本问题组件部分将会介绍这些方法的具体使用场景。下面开始分别讲述四种不同类型的问题组件。

文本问题组件

实现了问题组件的父类之后，接下来介绍如何实现一个具体的问题组件。首先以文本问题组件为例，文件存放在 `shared/question/shared/question-text` 目录下。示例代码如下：

```
// shared/question/shared/question-text/question-text.component.ts

import { Component, EventEmitter, Input, Output } from '@angular/core';

import { QuestionComponent } from '../..../index';
import { QuestionModel } from '../..../models/question.model';

@Component({
  selector: 'question-text',
  templateUrl: 'question-text.component.html'
})
export class QuestionTextComponent extends QuestionComponent {
  @Input() question: QuestionModel;
  @Input() editable: boolean;
  @Output() deleteQuestionRequest: EventEmitter<any> = new EventEmitter();
}
```

对应的模板文件示例代码如下：

```
<!-- shared/question/shared/question-text/question-text.component.html -->

<p>问答题</p>
<div *ngIf="!editable">
  <p>{{question.title}}</p>
  <input placeholder="请输入问题的答案" class="form-control" [(ngModel)]="question
    .answer" />
</div>

<div *ngIf="editable && isEditing">
  <input placeholder="请输入问题" class="form-control" [(ngModel)]="question.title
    " required />
  <div class="btns">
    <button (click)="onSave()" class="btn">保存</button>
    <button (click)="onCancel()" class="btn">取消</button>
  </div>
</div>

<div *ngIf="editable && !isEditing">
  <p>{{question.title}}</p>
  <input placeholder="请输入问题的答案" class="form-control" disabled="disabled" /
    >
  <div class="btns">
    <button (click)="onEdit()" class="btn btn-default">编辑</button>
    <button (click)="onDelete()" class="btn btn-default">删除</button>
  </div>
</div>
```

文本问题组件是最简单的问题组件，只需要继承问题组件父类，不需要做任何扩展。文本问题组件包含 `question` 和 `editable` 两个输入属性，它们的值都从问卷父组件中传入。同时文本问题组件还拥有删除问题的自定义事件 `deleteQuestionRequest`，该事件作为输出属性会被问卷父组件监听到。需要特别注明的是，这三个输入输出属性是所有问题组件共有的，因此在介绍其他类型的问题组件时将不再赘述。

接下来将讲解文本问题组件的模板文件，在代码中实现了问题在三种不同状态下的展示。

- 不可编辑状态：此时问卷处于回收或者已结束状态，在这两种状态下问题是不可编辑的，如图 15-3 所示。

问答题

问题标题

请输入问题的答案

编辑

删除

图 15-3 不可编辑状态

- 可编辑情况下的编辑状态：问卷作者正在编辑某个问题，在编辑过程中，可以随时保存或者取消编辑问题，如图 15-4 所示。

问答题

你希望通过实例学习到那些知识?

保存

取消

图 15-4 可编辑情况下的编辑状态

- 可编辑情况下的非编辑状态：问卷作者正在浏览该问题，在浏览过程中，可以随时编辑或删除问题，如图 15-5 所示。

问答题

如何评价 Angular 2.0 Final Release 的发布?

请输入问题的答案

编辑

删除

图 15-5 可编辑情况下的非编辑状态

最后添加文本问题组件的入口文件。示例代码如下：

```
// shared/question/shared/question-text/index.ts

export * from './question-text.component';
```

分值问题组件

分值问题组件和文本问题组件非常类似，它们的组件定义基本一致，区别只是组件元数据 selector 和 templateUrl 的取值及类名不同。分值问题组件的代码文件存放在 shared/question/shared/question-score 目录下，模板部分通过 range 类型的 <input> 控件来实现拉动选取值，在编辑状态下不可拖动，只展示分值效果。示例代码如下：

```

<!-- shared/question/shared/question-score/question-score.component.html -->

<p>分值题</p>

<div *ngIf="!editable">
  <p>{{question.title}}</p>
  <p class="range-field">
    <label>分值: {{question.answer}}</label>
    <input type="range" [(ngModel)]="question.answer" min="0" max="100" />
  </p>
</div>

<div *ngIf="editable && isEditing">
  <!-- ... -->
</div>

<div *ngIf="editable && !isEditing">
  <p>{{question.title}}</p>
  <p class="range-field">
    <label>分值: 50</label>
    <input type="range" value="50" min="0" max="100" disabled="disabled" />
  </p>
  <div class="btns">
    <button (click)="onEdit()" class="btn btn-default">编辑</button>
    <button (click)="onDelete()" class="btn btn-default">删除</button>
  </div>
</div>

```

单选问题组件

和文本、分值问题组件不同的是，单选问题组件多出了选择项功能，显示及处理逻辑也更加复杂。首先来看单选问题组件整体的定义，文件存放在 `shared/question/shared/question-radio` 目录下。组件示例代码如下：

```

// shared/question/shared/question-radio/question-radio.component.ts

// ...

@Component({
  selector: 'question-radio',
  templateUrl: 'question-radio.component.html'
})

```

```

})
export class QuestionRadioComponent extends QuestionComponent {

  @Input() question: QuestionModel;
  @Input() editable: boolean;
  @Output() deleteQuestionRequest: EventEmitter<any> = new EventEmitter();

  private key: number;

  ngOnInit() {
    this.copyQuestion();
    let options = this.question.options;
    this.key = options[options.length-1].key;
  }

  onDeleteOption(index:number) {
    if(this.question.options.length <= 1) {
      return;
    }
    this.question.options.splice(index, 1);
  }

  onAddOption() {
    this.question.options.push({key: ++this.key, value:'' });
  }
}

```

在单选问题组件类中，增加了一个数值类型的 `key` 属性，作为选项的唯一 ID。除此之外，单选问题组件类还增加了以下两个方法。

- `onAddOption()`：用于添加新的问题选项，每增加一个选项 `key` 值将加 1。
- `onDeleteOption()`：用于删除问题选项，当选项个数小于或等于 1 时，将不允许删除，以保证至少存在一个选项。

和文本、分值问题组件相比，单选问题组件的模板也有很大的不同，尤其是在编辑状态下。示例代码如下：

```

<!-- shared/question/shared/question-radio/question-radio.component.html -->

<p>单选题</p>

```

```

<div *ngIf="!editable">
  <!-- ... -->
</div>
<div *ngIf="editable && isEditing">
  <input placeholder="请输入问题" class="form-control" [(ngModel)]="question.title"
    required/>
  <div class="row pt-20" *ngFor="let option of question.options; let i=index">
    <div class="col-lg-10">
      <input placeholder="请填写选项" class="form-control" [(ngModel)]="option.
        value" type="text"/>
    </div>
    <div class="col-lg-2">
      <span *ngIf="question.options.length > 1" class="del-icon" (click)="
        onDeleteOption(i)">X</span>
    </div>
  </div>
  <div class="btns">
    <button (click)="onSave()" class="btn btn-default">保存</button>
    <button (click)="onAddOption()" class="btn btn-default">添加选项</button>
    <button (click)="onCancel()" class="btn btn-default">取消</button>
  </div>
</div>
<div *ngIf="editable && !isEditing">
  <p>{{question.title}}</p>
  <div *ngFor="let option of question.options" class="radio disabled">
    <label>
      <input name="group" type="radio" id="{{option.key}}" disabled="disabled">{{
        option.value}}
    </label>
  </div>
  <div class="btns">
    <button (click)="onEdit()" class="btn btn-default">编辑</button>
    <button (click)="onDelete()" class="btn btn-default">删除</button>
  </div>
</div>

```

和文本问题组件相比，单选问题组件的模板中新增了选项的展示、添加和删除操作，如图 15-6 所示。

图 15-6 单选问题组件

复选问题组件

复选问题组件和单选问题组件的定义非常相似，不同之处主要在于，复选问题组件支持选中多个选项，因此这里可以通过数组的形式将答案保存起来。复选问题组件的代码文件存放在 `shared/question/shared/question-checkbox` 目录下，与单选问题组件相比，主要区别在于增加了保存多个选项的 `setSelectedValue()` 方法。示例代码如下：

```
// shared/question/shared/question-checkbox/question-checkbox.component.ts
```

```
export class QuestionCheckboxComponent extends QuestionComponent {
  // ...
  ngOnInit(): void {
    this.copyQuestion();
    let options = this.question.options;
    this.key = options[options.length-1].key;
    if(!this.question.answer.selected){
      this.question.answer.selected = [];
    }
  }

  setSelectedValue(checked: boolean, value: string) {
    let selected = this.question.answer.selected;
    let index:number = selected.indexOf(value);
    if(checked){
      if(index < 0){
        selected.push(value);
      }
    }else{
      if(index > -1){

```



```

        selected.splice(index, 1);
    }
}
}
}

```

在复选问题组件中，我们为问题答案添加了一个数组类型的属性 `selected`，用来存储所选中的选项值。因此在初始化函数中，如果该属性值为空，则将被初始化为一个空数组。另外，在问卷回收状态下，点击复选框将触发 `setSelectedValue()` 方法，同时将选中状态和复选框对应的选项值作为传入参数，方法执行后，答案 `answer` 的 `selected` 属性将被更新。相关模板的部分代码如下：

```

<!-- shared/question/shared/question-checkbox/question-checkbox.component.html -->
<!-- ... -->
<div *ngFor="let option of question.options" class="radio">
  <label>
    <input name="group" type="checkbox" id="{{option.key}}"
      (click)="setSelectedValue($event.target.checked, option.value)">{{option.
        value}}
    </label>
  </div>
<!-- ... -->

```

由于复选问题组件和单选问题组件的模板代码非常类似，这里将不再详细列出。

接下来，将四类组件添加到问题共享模块（`QuestionSharedModule`）中，以便在其他模块中导入。示例代码如下：

```

// shared/question/shared/question-shared.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { QuestionCheckboxComponent } from './question-checkbox/index';
import { QuestionRadioComponent } from './question-radio/index';
import { QuestionScoreComponent } from './question-score/index';
import { QuestionTextComponent } from './question-text/index';

@NgModule({
  imports: [CommonModule, FormsModule],

```

```

    declarations: [
        QuestionCheckboxComponent,
        QuestionRadioComponent,
        QuestionScoreComponent,
        QuestionTextComponent
    ],
    exports: [
        QuestionCheckboxComponent,
        QuestionRadioComponent,
        QuestionScoreComponent,
        QuestionTextComponent,
        CommonModule,
        FormsModule
    ]
  })
  export class QuestionSharedModule { }

```

这里需要特别指出的是，上述代码中引入的 `FormsModule` 模块是必需的，这是因为在问题组件中用到了 `Input` 控件类型的 `ngModel` 属性，如果不引入该模块，代码将报错。

最后，把问题共享模块导入到问题模块中，使得整个问题模块的功能更加完善。示例代码如下：

```

// shared/question/question.module.ts

import { NgModule, ModuleWithProviders } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { QuestionSharedModule } from '../shared/question-shared.module';

@NgModule({
  imports: [CommonModule, FormsModule, QuestionSharedModule],
  exports: [CommonModule, QuestionSharedModule]
})
export class QuestionModule {

  static forRoot(): ModuleWithProviders {
    return {

```

```
        ngModule: QuestionModule
      };
    }
  }
}
```

15.2.3 问卷组件

组件开发

问卷组件是问卷编辑组件的子组件，同时又是问题组件的父组件，起到了衔接问卷编辑组件和问题组件的作用。和问题组件类似，问卷组件也会在多个地方被使用到，因此同样将该组件放到 `shared/questionnaire` 目录中。问卷组件的代码如下：

```
// shared/questionnaire/questionnaire.component.ts

import { Component, OnInit, EventEmitter, Input, Output } from '@angular/core';

import { QuestionnaireModel, QuestionnaireState } from '../models/questionnaire.model';

@Component({
  selector: 'questionnaire',
  styleUrls: ['questionnaire.component.css'],
  templateUrl: 'questionnaire.component.html'
})
export class QuestionnaireComponent implements OnInit {

  @Input() questionnaire: QuestionnaireModel;
  @Output() submitQuestionnaire = new EventEmitter();

  private editable: boolean;

  // ...

  ngOnInit() {
    this.editable = this.questionnaire && this.questionnaire.state ===
      QuestionnaireState.Created;
  }

  onDeleteQuestion(index: number) {
```

```

    this.questionnaire.questionList.splice(index, 1);
  }

  onSubmit() {
    this.submitQuestionnaire.emit(this.questionnaire);
  }
}

```

如上面的代码所示，问卷组件的基本内容包括：

- 定义输入变量 `questionnaire`，该变量指向从问卷编辑组件或其他父组件传递过来的问卷数据。
- 创建自定义事件 `submitQuestionnaire`，并在问卷提交时触发该事件，事件将以输出属性的形式传播给问卷编辑组件或其他父组件。
- 定义私有变量 `editable`，用来标记当前问卷是否可编辑，并且在组件初始化阶段给它赋值。
- 实现了由问题组件传递而来的删除事件 `deleteQuestionRequest`。

在本章的“数据模型”部分定义了问卷的三种状态：已创建状态、回收中状态和已结束状态。在初始化函数 `ngOnInit()` 中，在给变量 `editable` 赋值的时候可以看到，当问卷处于已创建状态时，变量值为 `true`，代表当前问卷是可编辑的；而当问卷处在回收中状态或者已结束状态时，变量值为 `false`，表示当前问卷是不可编辑的。在问卷组件的模板文件中，实现了在可编辑和不可编辑两种状态下的问卷展示。示例代码如下：

```

<!-- shared/questionnaire/questionnaire.component.html -->

<div *ngIf="questionnaire" class="questionnaire-container">
  <alert *ngIf="alert" [type]="alert.type">{{alert.msg}}</alert>
  <div class="questionnaire">
    <h1 *ngIf="!editable" class="text-center">{{questionnaire.title}}</h1>
    <h1 *ngIf="editable" class="text-center">
      <input placeholder="问卷标题" class="form-control" [(ngModel)]="questionnaire
        .title"/>
    </h1>
    <p *ngIf="!editable" class="text-center">{{questionnaire.starter}}</p>
    <p *ngIf="editable" class="text-center">
      <input placeholder="开头欢迎语" class="form-control" [(ngModel)]="
        questionnaire.starter"/>
    </p>
  </div>
</div>

```

```

<ul>
  <li *ngFor="let q of questionnaire.questionList; let i=index" [ngSwitch]="q.
    type">
    <label>第{{i+1}}题: </label>
    <div *ngSwitchCase="0">
      <question-text [question]="q" [editable]="editable" (
        deleteQuestionRequest)="onDeleteQuestion(i)"></question-text>
    </div>
    <div *ngSwitchCase="1">
      <question-radio [question]="q" [editable]="editable" (
        deleteQuestionRequest)="onDeleteQuestion(i)"></question-radio>
    </div>
    <div *ngSwitchCase="2">
      <question-checkbox [question]="q" [editable]="editable" (
        deleteQuestionRequest)="onDeleteQuestion(i)"></question-checkbox>
    </div>
    <div *ngSwitchCase="3">
      <question-score [question]="q" [editable]="editable" (
        deleteQuestionRequest)="onDeleteQuestion(i)"></question-score>
    </div>
  </li>
</ul>
<p *ngIf="!editable" class="text-center">{{questionnaire.ending}}</p>
<p *ngIf="editable" class="text-center">
  <input placeholder="结尾感谢语" class="form-control" [(ngModel)]="
    questionnaire.ending"/>
</p>
</div>
<div class="text-center">
  <button type="button" class="btn btn-primary" (click)="onSubmit()">提交</button
  >
</div>
</div>

```

在问卷可编辑状态下，问卷的标题、开始和结束语都是可以编辑的，同时也将问卷的可编辑状态传递给了问题组件，在上面的问题组件开发章节中已经使用到了它，不可编辑状态则相反。最后在模板中添加一个“提交”按钮，当问卷处于编辑状态时，单击“提交”按钮，所创建的问卷将被保存；当问卷处于发布状态时，单击“提交”按钮，问卷填写者的答案将被收集。

在问卷组件中使用到了四类问题组件，组件的引入是通过引入问题模块（QuestionModule）来完成的，对应的问卷模块文件示例代码如下：

```
// shared/questionnaire/questionnaire.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { QuestionModule } from '../question/question.module';
import { QuestionnaireComponent } from './questionnaire.component';

@NgModule({
  imports: [CommonModule, FormsModule, QuestionModule],
  declarations: [QuestionnaireComponent],
  exports: [QuestionnaireComponent, CommonModule, FormsModule]
})
export class QuestionnaireModule { }
```

添加问卷组件到问卷编辑页面中

在完成问卷组件的开发后，下一步就是将问卷组件添加到问卷编辑页面中。模板示例代码如下：

```
<!-- edit/edit.component.html -->

<div class="row edit-container">
  <div class="col-lg-3">
    <!-- ... -->
    <question-select (addQuestionRequest) = "onAddQuestion($event)"></question-select>
    <!-- ... -->
  </div>
  <div class="col-lg-9 questionnaire-container">
    <questionnaire [(questionnaire)]="questionnaire" (submitQuestionnaire)="onSubmitQuestionnaire($event)"></questionnaire>
  </div>
</div>
```

通过模板代码可以看到，在问卷标签上双向绑定了问卷属性 questionnaire，该属性的值将在问卷编辑组件中获取，同时还绑定了问卷的自定义提交事件 submitQuestionnaire。

接下来看如何在组件中实现这些功能。示例代码如下：

```
// edit/edit.component.ts

import { Component, OnInit } from '@angular/core';

import { QuestionType } from '../shared/models/question.model';
import { QuestionnaireModel } from '../shared/models/questionnaire.model';

@Component({
  selector: 'sd-edit',
  templateUrl: 'edit.component.html',
  styleUrls: ['edit.component.css']
})
export class EditComponent implements OnInit{
  private questionnaire:QuestionnaireModel;
  private id:string;

  // ...
}
```

最后在 Admin 模块中引入问卷模块。示例代码如下：

```
// admin.module.ts

// ...
import { QuestionnaireModule } from '../shared/questionnaire/questionnaire.module';

@NgModule({
  imports: [QuestionnaireModule /*...*/],
  // ...
})
export class AdminModule { }
```

至此，问卷组件已经被成功地添加到问卷编辑页面中。接下来，我们将要实现 EditComponent 组件中的部分方法。首先在构造函数中初始化一个空的问卷。示例代码如下：

```
// edit/edit.component.ts

// ...
constructor(private questionnaireService:QuestionnaireService){
```

```
this.questionnaire = {  
  title: '',  
  starter: '',  
  ending: '',  
  state: QuestionnaireState.Created,  
  questionList: []  
};  
}
```

目前问卷中的问题列表是空的，用户可以通过点击问题选择组件来添加问题，代码将处理该点击事件并将相应类型的问题项添加到问题列表中。事件处理的示例代码如下：

```
// edit/edit.component.ts  
  
// ...  
onAddQuestion(type: QuestionType) {  
  // 添加问题到问卷的问题列表中  
  switch(type){  
    case QuestionType.Text:  
    case QuestionType.Score:  
      this.questionnaire.questionList.push({  
        title: '问题标题',  
        type: type,  
        answer: ''  
      });  
      break;  
    case QuestionType.SingleSelect:  
      this.questionnaire.questionList.push({  
        title: '问题标题',  
        type: type,  
        options: [{ 'key': 0, 'value': '选项1' }],  
        answer: ''  
      });  
      break;  
    case QuestionType.MultiSelect:  
      this.questionnaire.questionList.push({  
        title: '问题标题',  
        type: type,  
        options: [{ 'key': 0, 'value': '选项1' }],
```



```
    answer: {}  
  });  
  break;  
  default:  
    break;  
}  
}
```

从上述代码中可以看出，和文本问题组件和分值问题组件相比，单选和多选问题组件初始化时增加了选项属性，且多选问题的答案被初始化为一个空对象。

至此，我们已经实现了点击问题选择组件中的问题类型，并在问卷组件中添加了相应问题的交互效果。最终页面展示如图 15-7 所示。

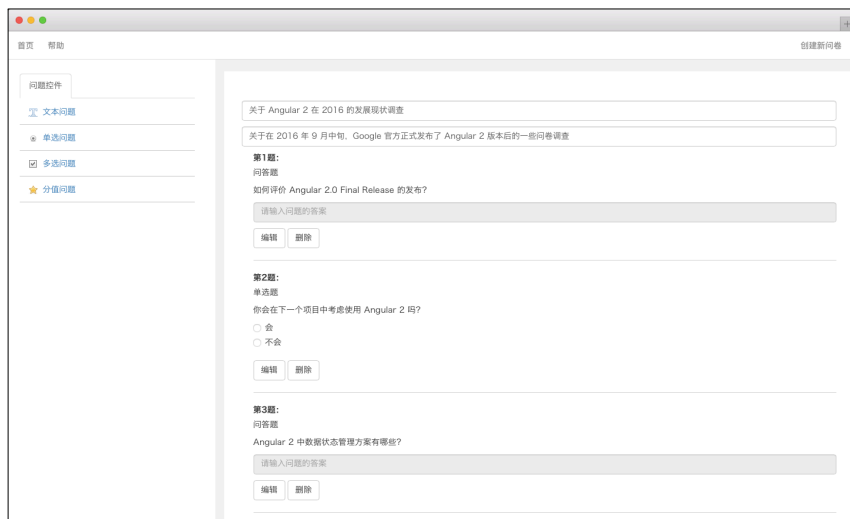


图 15-7 添加问题交互图

在实际的应用场景中，问卷并不总是被初始化为一个空对象的，例如在编辑一个已有问卷时，就会通过获取后台存储的数据初始化一个问卷对象。我们将获取后台数据的方法封装到一个问卷服务中，接下来的章节将介绍如何实现一个问卷服务，并在组件中使用它。

15.2.4 问卷服务

问卷服务的功能主要是和后端服务器完成数据交互。正如在第 14 章中所提到的目录布局划分方式，我们会将一些启动时只用到一次的指令、组件及全局单例的服务放在

Core 模块中，而 Core 模块最终将被根模块 AppModule 所加载。问卷服务将会在多个模块中使用到，所以我们选择将它作为一个全局单例的服务，放在 Core 模块中，并将其命名为 questionnaire.service.ts。其初始代码如下：

```
// core/services/questionnaire.service.ts

import { Injectable } from '@angular/core';
// ...

@Injectable()
export class QuestionnaireService {
  // ...
}
```

添加新问卷

首先需要在问卷服务类中实现添加新问卷的方法。它通过发送 POST 请求，将编辑好的问卷数据以 JSON 格式传递给后台保存，因此需要引入相关的 HTTP 服务及对应的数据模型等。示例代码如下：

```
// core/services/questionnaire.service.ts

import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Rx';
import { HttpClient } from '@angular/common/http';

import { QuestionnaireModel } from '../../shared/models/questionnaire.model';
import { SITE_HOST_URL } from '../../shared/index';

@Injectable()
export class QuestionnaireService {
  constructor(private http: HttpClient) {}

  private handleError(error) {
    console.error(error);
    return Observable.throw(error || 'server error');
  }

  // ...
}
```

```
// 添加新问卷
addQuestionnaire(questionnaire: QuestionnaireModel) {
  return this.http
    .post(SITE_HOST_URL + 'questionnaire/add', questionnaire)
    .catch(this.handleError);
}

// ...
}
```

代码中定义了一个通用的错误处理方法 `handleError()`，而在实际开发中，从错误监控的角度来说，一般会封装一个全局的错误上报服务，在可能出错的地方加上，用来进行监控与分析问题。

需要特别注意的是，添加新问卷的 `addQuestionnaire()` 方法返回的是一个 `Observable` 实例，获取到这个 `Observable` 实例之后可调用它的订阅函数 `subscribe()` 获取后端服务器返回的结果。更多关于 RxJS 的概念及相关操作符的介绍可参阅第 9 章。另外，代码中还引入了一个表示后台接口地址的 `SITE_HOST_URL` 常量，这个常量是从 `shared/config/env.config.ts` 文件中导出的，这个文件可以用于配置与全局相关的一些变量。示例代码如下：

```
// shared/config/env.config.ts

// ...
export const SITE_HOST_URL: string = 'http://localhost:5000/';
```

为了帮助读者更好地了解问卷服务，这里简单介绍添加新问卷的后台接口实现。在第 14 章准备的基础 `app.js` 文件中，添加如下代码：

```
// backend/app.js

// 添加新问卷
server.post('/questionnaire/add', (req, res) => {
  const item = req.body;
  item.id = uuid.v1();
  item.createDate = new Date().toLocaleDateString();
  db('questionnaires').push(item).then(() => {
    res.json({'success':true, data:item});
  });
});
```

后台接收到添加新问卷的请求后，首先获取请求传递过来的 body 对象，即待保存的问卷数据，然后给该对象添加唯一标识 ID 和创建时间，最后将该对象保存起来，并将保存后的问卷数据返回给前端应用。json-server 中使用的 lowdb 提供了类似于 find()、push() 等方法帮助我们从该数据文件中读写数据。在创建新的问卷时，使用了 node-uuid（可通过 npm install -save node-uuid 命令进行安装）为每个问卷生成一个唯一 ID。

正如前文所提及的，我们将问卷服务当作一个全局单例的服务，并注入到 Core 模块之中。在注入之前需要对服务进行注册：先引入服务，然后将该服务添加到模块的 providers 元数据中。完成服务的注册之后，接着需要把 CoreModule 导入到根模块 AppModule 之中，这样就可以在全局任意组件中使用问卷服务了。Core 模块的示例代码如下：

```
// core/core.module.ts

import { NgModule, Optional, SkipSelf } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule } from '@angular/router';

import { QuestionnaireService } from '../services/questionnaire.service';

@NgModule({
  imports: [CommonModule, RouterModule],
  providers: [QuestionnaireService]
})
export class CoreModule {
  constructor (@Optional() @SkipSelf() parentModule: CoreModule) {
    // 如果 Core 模块已经被导入过，即 parentModule 不为空时，抛出错误，防止重复导入
    if (parentModule) {
      throw new Error(
        'CoreModule is already loaded. Import it in the AppModule only');
    }
  }
}
```

注意到在上面代码的构造函数中，加入了防止 Core 模块被重复导入的异常提示，Core 模块只会在一开始时被导入到根模块中，不允许被其他模块重复导入。

在组件中使用问卷服务前，需要先在构造函数中注入该服务，因为上面已经在模块

中完成了 QuestionnaireService 服务的注册，可以直接在代码中使用这个服务。在问卷编辑组件中，在问卷提交方法里面会使用到问卷服务，示例代码如下：

```
// edit/edit.component.ts

// ...
import { QuestionnaireService } from '../core/services/questionnaire.service';
import { QuestionnaireModel } from '../shared/models/questionnaire.model';

// ...

constructor(private questionnaireService:QuestionnaireService) {
  // ...
}

// ...
onSubmitQuestionnaire(questionnaire: QuestionnaireModel) {
  // 保存问卷或回收答案
  if (questionnaire.state === QuestionnaireState.Created) {
    if (this.id && this.id !== '0') {
      // 编辑已有问卷
      this.questionnaireService.updateQuestionnaire(questionnaire)
        .subscribe(
          questionnaire => this.gotoCenter(),
          error => console.log(error));
    } else {
      // 创建新问卷
      this.questionnaireService.addQuestionnaire(questionnaire)
        .subscribe(
          questionnaire => this.gotoCenter(),
          error=> console.error(error)
        );
    }
  }
}

// ...
```

上文提到过，当问卷处于已创建状态时，如果存在问卷 ID，则是编辑一个已有问卷；否则创建一个新问卷。

到这里为止，添加新问卷的功能就已经开发完成了，用户编辑完问卷，提交成功后，可以在 `backend/_db.json` 中查看保存后的问卷数据。

获取已有问卷

在问卷编辑组件的初始化函数中，已经给问卷数据赋了初始值。但实际上，当编辑一个已有问卷时，问卷数据是可以获取到的。为了获取该数据，首先要读取到 URL 中的问卷 ID，然后再去调用问卷服务的相关方法，这里的问卷 ID 可以通过路由参数得到。

在问卷编辑组件中，引入 `ActivatedRoute` 路由服务，并在构造函数中注入它。修改后的部分相关代码如下：

```
// edit/edit.component.ts

// ...
import { Router, ActivatedRoute } from '@angular/router';

@Component({
  selector: 'sd-edit',
  templateUrl: 'edit.component.html',
  styleUrls: ['edit.component.css']
})
export class EditComponent implements OnInit {

  private questionnaire: QuestionnaireModel;
  private id: string;

  constructor(private questionnaireService: QuestionnaireService,
    private activatedRoute: ActivatedRoute, private router: Router) {

    // 初始化一个空的问卷对象
    this.questionnaire = {
      title: '',
      starter: '',
      ending: '',
      state: QuestionnaireState.Created,
      questionList: []
    };
  }
}
```

```

ngOnInit() {

  // 初始化问卷数据
  this.id = this.activatedRoute.snapshot.params['id'];

  if(this.id && this.id !== '0'){

    // ID存在, 代表当前页面为编辑已有问卷页面, 调用服务获取问卷对象信息
    this.questionnaireService.getQuestionnaireById(this.id)
      .subscribe(
        questionnaire => this.questionnaire = questionnaire,
        error => console.log(error)
      );
  }
}
}

```

在问卷编辑组件中, 首先通过 `ActivatedRoute` 服务获取到 URL 传递过来的问卷 ID。如果 ID 为空, 则表示当前为创建新问卷的页面, 会初始化一个空的问卷数据; 反之, 则为编辑已有问卷的页面, 可以通过调用问卷服务 `QuestionnaireService` 的 `getQuestionnaireById()` 方法获取到相应的问卷数据。问卷服务的相关代码如下:

```

// core/service/questionnaire.service.ts

// 根据 ID 获取问卷信息
getQuestionnaireById(id: string) {
  return this.http.get(SITE_HOST_URL + 'questionnaire/' + id)
    .map(res => <QuestionnaireModel>res.json().data)
    .catch(this.handleError);
}

```

后台接口代码如下:

```

// backend/app.js

// 根据 ID 获取问卷数据
server.get('/questionnaire/:id', (req, res) => {
  const questionnaire = db('questionnaires').find({id: req.params.id});
  res.json({'success':true, data:questionnaire});
});

```

最后，在问卷服务中补充完整更新已有问卷的代码。示例代码如下：

```
// core/services/questionnaire.service.ts

// 更新已有问卷
updateQuestionnaire(questionnaire:QuestionnaireModel) {
  let body = JSON.stringify(questionnaire);
  let headers = new Headers({'Content-Type':'application/json'});
  let options = new RequestOptions({headers:headers});

  return this.http.post(SITE_HOST_URL + '/questionnaire/update', body, options)
    .map(res => <QuestionnaireModel>res.json().data)
    .catch(this.handleError);
}
```

后台接口更新已有问卷的示例代码如下：

```
// backend/app.js

// 更新已有问卷数据
server.post('/questionnaire/update', (req, res) => {
  const item = req.body;
  db('questionnaires').chain().find({id:item.id}).assign(item).value();
  res.json({'success':true, data:item});
});
```



我们还可以改造 Angular 自带的 `HttpService` 服务，以满足类似于统一的错误处理、回调捕获、加载动画等功能需求，具体实现请参照第 9 章服务与 RxJS 章节。

15.2.5 问卷大纲组件

至此，问卷编辑页面已经完成了绝大部分组件的开发，最后将讲解问卷大纲组件的实现过程。

组件开发

在问卷编辑页面中，问卷大纲用于展示当前问卷中已有问题的标题列表，帮助问卷创建者更方便、直观地了解问卷的情况。和问题选择组件类似，问卷大纲组件同样只有

在问卷编辑页面中才会被使用到，该组件也将被放到与问题选择组件同级的目录下。示例代码如下：

```
// edit/shared/question-outline/question-outline.component.ts

import { Component, Input } from '@angular/core';

import { QuestionnaireModel } from '../../../shared/models/questionnaire.model';

@Component({
  selector: 'questionnaire-outline',
  styleUrls: ['questionnaire-outline.component.css'],
  templateUrl: 'questionnaire-outline.component.html'
})
export class QuestionnaireOutlineComponent {
  @Input() questionnaire: QuestionnaireModel;
}
```

接下来创建问卷大纲组件的模板文件。示例代码如下：

```
<!-- edit/shared/question-outline/question-outline.component.html -->

<ul *ngIf="questionnaire" class="questionnaire-outline">
  <li *ngFor="let q of questionnaire.questionList; let i=index">
    <span>{{(i + 1) + ". " + q.title}}</span>
  </li>
</ul>
```

类似于问题选择组件，我们也将问卷大纲组件添加到 EditSharedModule 模块中。示例代码如下：

```
// edit/shared/edit-shared.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { QuestionSelectComponent } from '../question-select/index';
import { QuestionnaireOutlineComponent } from '../questionnaire-outline/index';

@NgModule({
  imports: [CommonModule],
  declarations: [QuestionSelectComponent, QuestionnaireOutlineComponent],
```

```

    exports: [QuestionSelectComponent, QuestionnaireOutlineComponent, CommonModule]
  })
  export class EditSharedModule { }

```

添加问卷大纲组件到问卷编辑页面中

这里将使用选项卡来控制问题选择和问卷大纲面板的展示，我们将从 ngx-bootstrap 样式库中把 TabsModule 导入到 EditModule 模块之中，接着就可以在模板中使用选项卡样式了。示例代码如下：

```

// admin.module.ts

// ...
import { TabsModule } from 'ngx-bootstrap';

@NgModule({
  imports: [TabsModule /*...*/]
  // ...
})
export class AdminModule { }

```

同时修改问卷编辑组件的模板代码如下：

```

<!-- edit/edit.component.html -->

<div class="row edit-container">
  <div class="col-lg-3">
    <div class="sidebar">
      <tabset type="tabs">
        <tab heading="问题控件">
          <question-select (addQuestionRequest)="onAddQuestion($event)"></question-select>
        </tab>
        <tab heading="问卷大纲">
          <questionnaire-outline [questionnaire]="questionnaire"></questionnaire-outline>
        </tab>
      </tabset>
    </div>
  </div>
  <!-- ... -->
</div>

```

问卷大纲组件添加完成后，问卷编辑页面中的问卷大纲展示效果如图 15-8 所示。

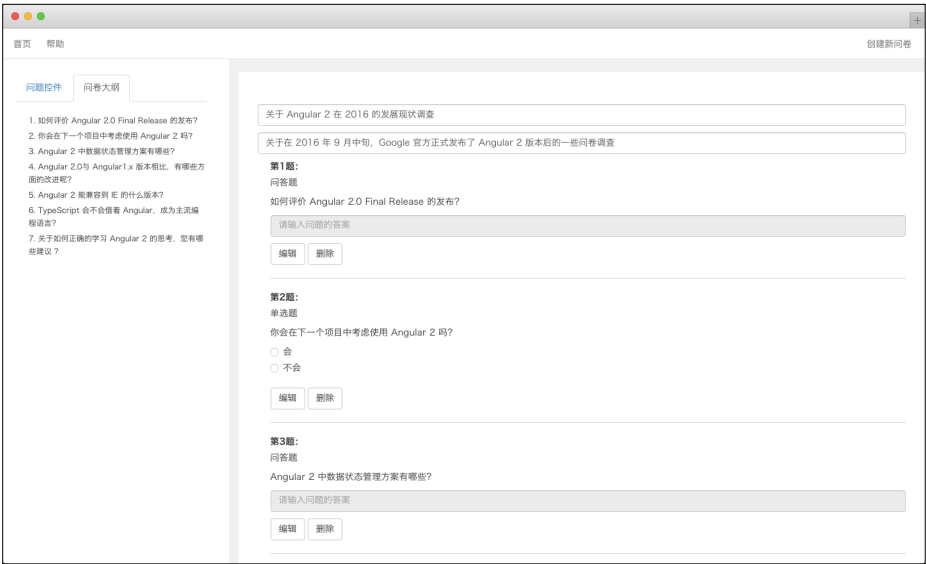


图 15-8 问卷大纲展示效果

15.3 小结

本章首先整体介绍了问卷编辑页面的功能设计及数据模型，接着通过添加问题选择组件、问卷、问卷大纲等子组件逐步完善了页面的功能，最后介绍了如何通过问卷服务完成和后台的数据交互功能。

通过一个完整的问卷编辑功能的开发，相信读者对 Angular 基本的开发流程和核心功能的使用已经有了更为直观的认识。除了问卷编辑模块，问卷调查系统还包括问卷列表和问卷发布等功能，这些功能的开发涉及组件开发、模板绑定、数据服务等方面的知识，我们将在下一章中介绍，以帮助读者进一步熟悉 Angular 的开发。

16

我的问卷模块

在第 15 章中我们完成了问卷编辑模块的开发，本章将继续完成我的问卷模块的开发。

在我的问卷页中，用户可以通过问卷列表组件查看所有问卷的标题、状态和创建时间，也可以点击选中某个具体的问卷列表项并在问卷详情板块中查看当前问卷的更多信息。除此之外，用户还可以在问卷操作控件组中执行问卷的预览、编辑、发布、回收及删除等操作。接下来看看我的问卷模块的组件设计，如图 16-1 所示。

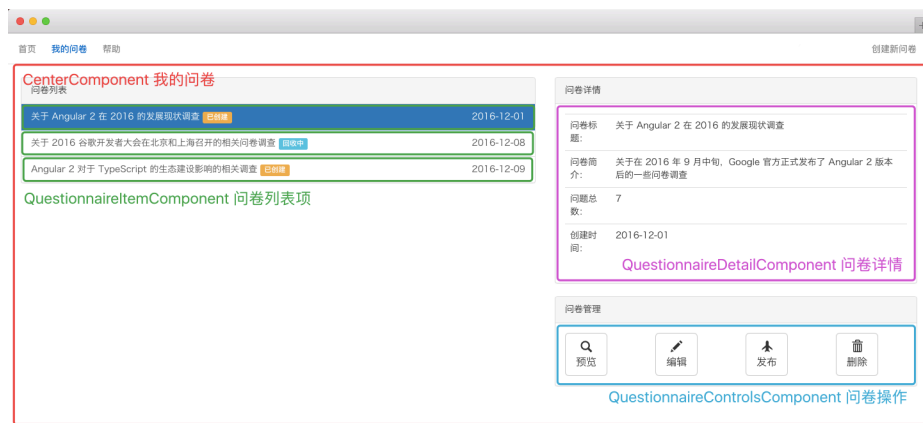


图 16-1 我的问卷模块的组件设计

根据图 16-1 所示的设计，下面将开始创建我的问卷模块目录。与问卷编辑模块类似，我的问卷也单独作为一个模块，添加相关文件并配置我的问卷模块的路由链接到导航栏中，这里将文件统一存放到 `app/admin/center` 目录中，目录结构如下：

```
app/admin/center
├── center.component.css
├── center.component.html
├── center.component.ts
├── index.ts
├── shared
│   ├── center-shared.module.ts
│   ├── index.ts
│   ├── questionnaire-controls
│   │   ├── index.ts
│   │   ├── questionnaire-controls.component.css
│   │   ├── questionnaire-controls.component.html
│   │   └── questionnaire-controls.component.ts
│   ├── questionnaire-detail
│   │   ├── index.ts
│   │   ├── questionnaire-detail.component.html
│   │   └── questionnaire-detail.component.ts
│   └── questionnaire-item
│       ├── index.ts
│       ├── questionnaire-item.component.html
│       └── questionnaire-item.component.ts
```

其中，该模块下的 `shared` 目录用于存放我的问卷模块的子组件，其他外层文件（如 `center.routes.ts` 等）的代码和编辑模块的十分类似，这里不再赘述。接下来主要讲解如何实现 `shared` 目录下的相关子组件。

16.1 问卷列表

问卷列表展示了用户创建的所有问卷集合，每个列表项对应的是下面要讲的问卷列表项组件。

16.1.1 问卷列表项

问卷列表项组件（`QuestionnaireItemComponent`）展示了问卷标题、问卷状态和问卷创建时间等内容，相关模板的示例代码如下：

```
<!-- center/shared/questionnaire-item/questionnaire-item.component.html -->
```

```
{{questionnaire.title}}
<span class="label" [ngClass]="stateClass">{{stateText}}</span>
<span class="pull-right">{{questionnaire.createDate}}</span>
```

在模板代码中使用内置指令 `ngClass` 为标签绑定了在不同问卷状态下对应的样式类 (`stateClass`)，同时也将问卷状态以文本的形式 (`stateText`) 展示出来。接着在组件的初始化函数中给它们赋值，示例代码如下：

```
// center/shared/questionnaire-item/questionnaire-item.component.ts

import { Component, OnInit, OnChanges, SimpleChanges, Input } from '@angular/core';

import { QuestionnaireModel, QuestionnaireState } from '../../shared/models/questionnaire.model';

@Component({
  selector: 'questionnaire-item',
  templateUrl: 'questionnaire-item.component.html'
})
export class QuestionnaireItemComponent implements OnInit, OnChanges {

  @Input() questionnaire: QuestionnaireModel;

  private stateText: String;
  private stateClass: String;

  ngOnChanges(changes: SimpleChanges) {
    let questionnaireChange = changes['questionnaire'];
    if(questionnaireChange.previousValue.state &&
        questionnaireChange.currentValue.state !== questionnaireChange.
            previousValue.state){
      this.questionnaire = changes['questionnaire'].currentValue;
      this.setState();
    }
  }

  ngOnInit() {
    this.setState();
  }
}
```

```

    }

    setState() {
      switch(this.questionnaire.state){
        case QuestionnaireState.Created:
          this.stateText = '已创建';
          this.stateClass = 'label-warning';
          break;
        case QuestionnaireState.Published:
          this.stateText = '回收中';
          this.stateClass = 'label-info';
          break;
        case QuestionnaireState.Finished:
          this.stateText = '已结束';
          this.stateClass = 'label-success';
          break;
        default:
          break;
      }
    }
  }
}

```

在上述代码中，问卷列表项组件定义了一个问卷输入变量 `questionnaire`，用来接收问卷列表传递过来的问卷信息。同时还使用内置指令 `ngClass` 绑定一个组件的成员变量 `stateClass` 给标签添加了一个样式类。如果需要为标签添加动态样式类，则可以通过为 `ngClass` 绑定一个函数方法来实现，读者可以参考第 7 章模板章节中关于内置指令部分的内容。

16.1.2 显示问卷列表

介绍完问卷列表项组件之后，我们将在我的问卷模块中通过服务获取问卷列表数据并把列表显示出来。我的问卷组件改动的示例代码如下：

```

// center/center.component.ts

// ...
export class CenterComponent implements OnInit {
  private questionnaires:QuestionnaireModel[] = [];
  private selectedQuestionnaire:QuestionnaireModel;
  private selectedIndex:number;

```

```

private isEmpty:boolean;

constructor(private questionnaireService:QuestionnaireService) { }

ngOnInit() {
  this.questionnaireService.getQuestionnaires()
    .subscribe(
      questionnaires => {
        // 后端返回空对象或者空的问卷数组
        if(!questionnaires || questionnaires.length === 0){
          this.isEmpty = true;
          return;
        }
        this.isEmpty = false;
        this.questionnaires = questionnaires;
        this.selectedQuestionnaire = this.questionnaires[0];
        this.selectedIndex = 0;
      },
      error => console.error(error)
    );
}
}

```

在 `CenterComponent` 组件的初始化钩子函数 `ngOnInit()` 中，调用问卷服务获取问卷列表。如果返回值为空对象或者空数组，则设置 `isEmpty` 属性值为 `true`，否则该属性值为 `false`，同时将当前选中的问卷（`selectedQuestionnaire`）设置为问卷数组的第一个元素，并且标记当前选中元素的索引值（`selectedIndex`）为 0。

获取到问卷列表的数据后，接下来在模板中展示出来。示例代码如下：

```

<!-- center/center.component.html -->

<!-- ... -->
<div class="row center-container" *ngIf="!isEmpty">
  <div class="col-lg-7">
    <div class="panel panel-default">
      <div class="panel-heading">问卷列表</div>
      <div class="list-group">
        <a class="list-group-item questionnaire-item" *ngFor="let item of
          questionnaires;let i=index;"

```



```

        (click)="onSelect(item, i)" [class.active]="item===selectedQuestionnaire"
      >
        <questionnaire-item [questionnaire]="item"></questionnaire-item>
      </a>
    </div>
  </div>
</div>
<!-- ... -->
</div>

```

问卷列表除展示问卷的基本信息之外，还需要实现点击选中当前问卷的功能，并且给当前问卷添加激活状态下的 `active` 样式类。在上面的代码中，分别通过添加响应点击处理事件的 `onSelect()` 方法和 CSS 类绑定的方法实现了这两个功能。

注意上面的模板提供了不同于内置指令 `ngClass` 的另一种绑定样式类的方法，即 CSS 类绑定。有关 CSS 类绑定的详细介绍可以参考第 7 章模板章节中的属性绑定部分。另外，响应点击选中当前问卷事件的 `onSelect()` 方法的实现代码如下：

```

// center/center.component.ts

// ...
export class CenterComponent implements OnInit {
  // ...
  onSelect(questionnaire:QuestionnaireModel, index:number) {
    this.selectedQuestionnaire = questionnaire;
    this.selectedIndex = index;
  }
}

```

16.1.3 显示问卷详情

问卷详情组件（`QuestionnaireDetailComponent`）展示了当前所选中问卷的更多信息，包括问卷标题、问卷简介、问题总数、创建时间和问卷公开链接。问卷详情组件的实现比较简单，只需要将问卷列表中所选中的问卷项作为输入属性传入，并在子组件中将问卷信息展示出来即可。在我的问卷模板中引入问卷详情组件的示例代码如下：

```

<!-- center/center.component.html -->

<!-- ... -->
<div class="row center-container" *ngIf="!isEmpty">
  <!-- ... -->

```

```

<div class="col-lg-5">
  <div class="panel panel-default">
    <div class="panel-heading">问卷详情</div>
    <div class="panel-body">
      <questionnaire-detail [questionnaire]="selectedQuestionnaire"></
        questionnaire-detail>
    </div>
  </div>
<!-- ... -->
</div>
</div>

```

接下来在问卷详情组件中定义一个输入属性变量 `questionnaire`，用于接收父组件传递过来的当前问卷的相关信息。示例代码如下：

```

// center/shared/questionnaire-detail/questionnaire-detail.component.ts

// ...
export class QuestionnaireDetailComponent {
  @Input() questionnaire: QuestionnaireModel;
}

```

需要注意的是，在展示问卷详情时，首先会判断问卷数据是否为空，因为传入数据为空可能会导致模板转换错误，这也是在开发过程中容易被忽略的。示例代码如下：

```

<!-- center/shared/questionnaire-detail/questionnaire-detail.component.html -->

<!-- 判断变量是否为空 -->
<table *ngIf="questionnaire" class="table">
  // ...
  <tr>
    <td>问题总数: </td>
    <td>{{questionnaire.questionList.length}}</td>
  </tr>
  <tr>
    <td>创建时间: </td>
    <td>{{questionnaire.createDate}}</td>
  </tr>
  <tr *ngIf="questionnaire.state===1">
    <td colspan="2">
      <a [routerLink]="['/published', questionnaire.id]">问卷发布链接</a>
    </td>
  </tr>

```

```
        </td>
      </tr>
</table>
```

16.2 问卷操作

问卷操作组件（QuestionnaireControlsComponent）封装了问卷的预览、编辑、发布、回收及删除等操作。和问卷详情组件类似，也是将问卷列表中所选中的问卷项作为输入属性传递给问卷操作组件的。在我的问卷模板中引入问卷操作组件的示例代码如下：

```
<!-- center/center.component.html -->

<div class="row center-container" *ngIf="!isEmpty">
  <!-- ... -->
  <div class="col-lg-5">
    <!-- ... -->
    <div class="panel panel-default">
      <div class="panel-heading">问卷管理</div>
      <div class="panel-body">
        <questionnaire-controls [questionnaire]="selectedQuestionnaire"
          (deleteQuestionnaire)="onDeleteQuestionnaire()"
          (publishQuestionnaire)="onPublishQuestionnaire()"></questionnaire-
            controls>
      </div>
    </div>
  </div>
</div>
```

问卷在不同状态下，可执行的操作是不同的。例如所有的问卷都可以执行预览和删除操作，但是只有已创建状态下的问卷才可以编辑和发布。另外，回收中的问卷还可以结束回收、查看当前的统计信息，而已结束的问卷则可以将统计结果导出。它们的对应关系如表 16-1 所示。

表 16-1 对不同状态下的问卷可执行的操作

问卷状态	可执行的操作
已创建	预览、编辑、发布和删除
回收中	预览、结束、统计和删除
已结束	预览、统计、导出和删除

在问卷的所有可执行操作中，问卷编辑和问卷删除功能最简单，这里就不再展开介绍了。而问卷统计和数据导出的功能，也不在当前实战项目的实现范围内，所以下面将主要介绍问卷的预览和发布功能。

问卷预览功能可以让问卷创建者提前看到问卷发布后的展示形态。而问卷发布功能会生成一个问卷的公开链接，并将问卷状态设置为回收中，问卷的公开链接将公开给问卷填报者访问。这两个功能都涉及发布后的问卷页面，接下来就先实现发布后的问卷页面。

16.2.1 发布后的问卷页面

为了方便讲解，同样将发布后的问卷页面模块集成到同一个 Angular 应用中。



在实际的项目开发中，由于发布后的问卷页面通常只是一个简单的静态页面，无须采用 SPA（单页应用）方式来构建，更好的做法是将问卷发布页面抽离出来作为单独项目，并且采用后端直出的方式，以提升用户体验。

添加 `published` 目录并创建相关文件，目录结构如下：

```
app
├── published
│   ├── published.component.html
│   ├── published.component.ts
│   ├── published.module.ts
│   ├── published-routing.module.ts
│   ├── published.routes.ts
│   └── index.ts
```

因为发布后的问卷页面不属于 `admin` 路由的范畴，因此将直接添加到根路由模块中。接下来主要看该页面的实现，其组件模板代码如下：

```
<!-- published/published.component.html -->

<div class="published-container">
  <questionnaire [(questionnaire)]="questionnaire"></questionnaire>
</div>
```

从代码中可以看出，发布后的问卷页面复用了问卷组件 <questionnaire>，第 15 章已经介绍过发布后的问卷组件的展示，这里就不再赘述了。不同之处在于，预览中的问卷页面是不需要提交到后台的，所以这里需要将按钮功能从“提交”替换为“返回”，单击后返回到我的问卷页面。与按钮相关的模板代码修改如下：

```
<!-- shared/questionnaire/questionnaire.component.html -->

<div *ngIf="questionnaire" class="questionnaire-container">
  <!-- ... -->
  <div class="text-center">
    <button type="button" class="btn btn-primary" (click)="onSubmit()">{{
      isPreviewPage ? "返回" : "提交"}}</button>
  </div>
</div>
```

如果当前页是预览页面，则会在 URL 上添加 type=preview 这个 Query 参数，以此代表问卷预览页面。在 Angular 中，对 Query 参数的获取，需要借助于 ActivatedRoute 服务的 queryParams 属性来完成，它是一个 Observable 对象。示例代码如下：

```
// shared/questionnaire/questionnaire.component.ts

// ...
import { Router, ActivatedRoute } from '@angular/router';
// ...
export class QuestionnaireComponent implements OnInit {

  // ...
  private isPreviewPage: boolean;

  constructor(private router: Router, private activatedRoute: ActivatedRoute){}

  ngOnInit() {
    // ...
    // 判断当前 URL 是否为预览页面
    this.activatedRoute.queryParams.subscribe(params => {
      this.isPreviewPage = params['type'] === 'preview';
    });
  }
  // ...
}
```

在之前的按钮点击事件中，只处理了在编辑状态下提交问卷的情况。除此之外，还有两种情况需要处理，一种是预览页面，单击该按钮将返回到我的问卷页面；另一种是发布后的问卷页面，用户在填写完毕后单击该按钮，这时候需要将答案保存起来。由于问卷统计的功能并不包含在这个项目的实现范围内，所以在这种情况下，会在用户提交问卷后在页面的顶部给出提示。修改后的按钮点击处理函数如下：

```
// shared/questionnaire/questionnaire.component.ts

// ...
onSubmit() {
  if (this.isPreviewPage) {
    this.router.navigateByUrl('admin/center');
    return;
  }

  switch (this.questionnaire.state) {
    case QuestionnaireState.Created:
      this.submitQuestionnaire.emit(this.questionnaire);
      break;
    case QuestionnaireState.Published:
      this.alert = {
        type: 'success',
        msg: '提交成功，感谢您的耐心回答。'
      };
      break;
    default:
      break;
  }
}
// ...
```

由于 AlertModule 已经注入到 QuestionnaireModule 模块中，提示功能直接使用了 ngx-bootstrap 中的 Alert 组件（由问卷组件类的变量 alert 控制），发布后的问卷页面如图 16-2 所示。

关于 Angular 2 在 2016 的发展现状调查

关于在 2016 年 9 月中旬, Google 官方正式发布了 Angular 2 版本后的一些问卷调查

第1题:
问答题
如何评价 Angular 2.0 Final Release 的发布?

好

第2题:
单选题
你会在下一个项目中考虑使用 Angular 2 吗?

☒ 会
☐ 不会

第3题:
问答题
Angular 2 中数据状态管理方案有哪些?

请输入问题的答案

第4题:
分值题
Angular 2.0 与 Angular 1.x 版本相比, 有哪些方面的改进呢?

分值:

第5题:
复选题

图 16-2 发布后的问卷页面

16.2.2 问卷操作组件

最后要介绍的就是问卷操作组件, 它可以对问卷进行管理操作, 其模板文件的核心代码如下:

```
<!-- center/shared/questionnaire-controls/questionnaire-controls.component.html -->

<div class="row" *ngIf="questionnaire">
  <div class="col-lg-3">
    <button type="button" class="btn btn-default btn-lg" (click)="onPreview()">
      <span class="glyphicon glyphicon-search"></span>
      <span class="control-text">预览</span>
    </button>
  </div>
  <div class="col-lg-3" *ngIf="questionnaire.state===0">
    <button type="button" class="btn btn-default btn-lg" (click)="onPublish()">
      <span class="glyphicon glyphicon-plane"></span>
      <span class="control-text">发布</span>
    </button>
  </div>

  <!-- 编辑、结束发布等其他操作按钮 -->
  <!-- ... -->
</div>
```

上述模板对应的预览和发布功能在组件内部的实现代码如下：

```
// center/shared/questionnaire-controls/questionnaire-controls.component.ts

// ...
onPreview(){
  this.router.navigateByUrl('published/' + this.questionnaire.id + '?type=preview');
}

onPublish(){
  this.publishQuestionnaire.emit();
}
// ...
```

问卷预览就是直接跳转到发布后的问卷页面，同时带上 `type=preview` 参数用来标记当前页为预览页面，从而控制底部按钮的显示和点击逻辑。问卷发布则触发问卷发布事件 `publishQuestionnaire`，并将该事件传递到父组件（我的问卷组件）中做相应处理。示例代码如下：

```
<!-- center/center.component.html -->

<!-- ... -->
<questionnaire-controls [questionnaire]="selectedQuestionnaire"
                        (deleteQuestionnaire)="onDeleteQuestionnaire()"
                        (publishQuestionnaire)="onPublishQuestionnaire()"
                        (endQuestionnaire)="onEndQuestionnaire()">
</questionnaire-controls>
<!-- ... -->
```

从上述模板代码中可以看到，除发布事件外，问卷的删除和结束回收的事件也会传递过来，其中在我的问卷组件中发布事件处理方法的示例代码如下：

```
// center/center.component.ts

// ...
onPublishQuestionnaire(){
  this.questionnaireService.updateQuestionnaireState(this.selectedQuestionnaire.
    id, QuestionnaireState.Published)
    .subscribe(
      questionnaire => {
        this.selectedQuestionnaire.state = QuestionnaireState.Published;
        this.questionnaires[this.selectedIndex] = Object.assign({}, this.
```



```

        selectedQuestionnaire);
    },
    error => console.log(error)
  );
}
// ...

```

在发布事件的回调方法中调用了问卷服务的 `updateQuestionnaireState()` 方法来更新问卷状态，将当前问卷状态置为回收中，并在服务调用成功后更新所选中问卷的状态。需要特别注意的是，问卷的状态修改成功后，还需要同步更新问卷列表中对应问卷的状态。换句话说，需要在问卷列表项组件中监听问卷的状态是否有改变，这里通过组件的生命周期钩子 `OnChanges()` 来实现，示例代码如下：

```

// center/shared/questionnaire-item/questionnaire-item.component.ts

import { Component, OnInit, OnChanges, SimpleChanges, Input } from '@angular/core';

import { QuestionnaireModel, QuestionnaireState } from '../../shared/models/questionnaire.model';

// ...
export class QuestionnaireItemComponent implements OnInit, OnChanges {

  @Input() questionnaire: QuestionnaireModel;

  private stateText: String;
  private stateClass: String;

  ngOnChanges(changes: SimpleChanges){
    let questionnaireChange = changes['questionnaire'];
    if(questionnaireChange.previousValue.state &&
        questionnaireChange.currentValue.state !== questionnaireChange.
            previousValue.state){
      this.questionnaire = changes['questionnaire'].currentValue;
      this.setState();
    }
  }

  // ...
}

```

实现 OnChanges() 钩子后，Angular 便可以监听输入属性 questionnaire 的变化，从而触发 ngOnChanges 事件，该事件接收一个 SimpleChanges 类型的参数，以获取属性变化前（previousValue）和变化后（currentValue）的值。当问卷的状态值发生变更时，则会更新问卷的显示。关于变化监测的更多内容，可以参阅第 6 章组件章节中的介绍。

至此，整个我的问卷页面开发完毕，最终展示效果如图 16-3 所示。

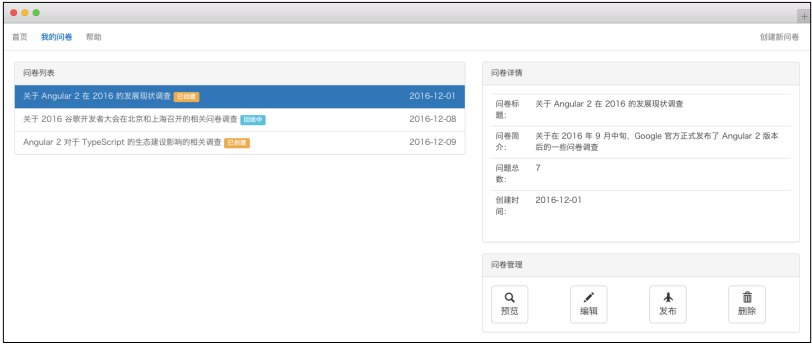


图 16-3 我的问卷页面最终展示效果

16.3 小结

本章主要实现了我的问卷模块。首先介绍了问卷列表和问卷详情部分的开发细节，了解了样式类绑定等相关知识；接着介绍了因为引入问卷发布页面而引起的路由配置更改的需求，从而更深入地了解了路由和子路由的概念；最后实现了问卷的预览和发布操作，展示了如何使用变化监测等功能。

17

用户管理模块

在第 16 章中我们完成了我的问卷模块的开发，本章将继续完成用户管理模块的开发。

在正常项目开发中，前面实现的一些模块都需要依赖用户管理功能。用户管理功能是系统最基础的功能之一，在很多系统中都是必不可少的一部分，具有一定的代表性。同时，用户管理涉及的表单操作和表单项输入验证也是展示 Angular 强大的表单处理能力的好机会。接下来将重点讲述用户管理模块中的用户注册、用户登录及认证授权功能。

以用户注册为例，注册页的组件设计如图 17-1 所示。

图 17-1 用户注册页的组件设计

在实际项目中，用户注册功能相对比较简单，涉及的表单项也不复杂，一般流程如图 17-2 所示。

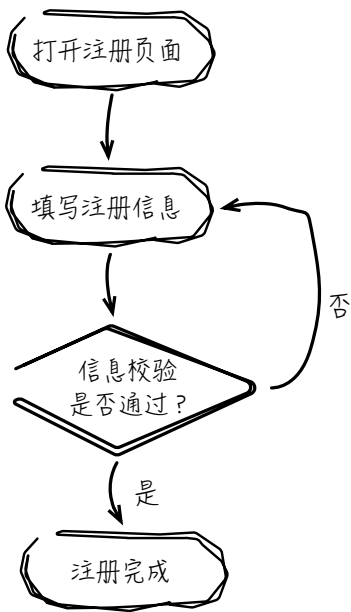


图 17-2 用户注册流程图

17.1 开发简单注册页

先抛开校验和代码重用等问题，从一个最简单的注册页说起，模板的示例代码如下：

```
<!-- user/shared/register/register.component.html -->

<form (ngSubmit)="register()">
  <label for="username">用户名: </label>
  <input name="username" id="username" [(ngModel)]="username">
  <label for="password">密码: </label>
  <input type="password" name="password" id="password" [(ngModel)]="password">
  <input type="submit" value="提交">
</form>
```

在上面的模板代码中，都是一些简单的数据绑定的内容，接下来看看其对应的组件实现，示例代码如下：

```
// user/shared/register/register.component.ts

import { Injectable } from '@angular/core';

@Component({
  selector: 'register',
  templateUrl: 'register.component.html'
});
export class RegisterComponent {
  username: string;
  password: string;
  constructor() {}
  register () {
    // 注册逻辑
  }
}
```

这个用户注册示例已经对代码做了精简，且没有加入样式部分的代码，可以看出与传统的开发方式（如 jQuery）相比还是有类似的地方的，但区别也很明显，主要区别如下：

- 代码更简练。
- 相关的代码都放在一个文件夹里面，更有利于代码的维护。
- 整个注册功能可以独立组成一个组件，便于复用。

在本书的实战项目中，注册功能可能只用一次就够了，不需要考虑复用。但是，在实际项目中，显然不会这么简单，从表单的角度来讲，表单在一个网站中可能会多次重复出现，对于表单的每一个控件来讲，都存在复用的可能，所以将表单控件做成一个组件就显得很有必要了。

表单由处理用户输入的各类控件组成，常见的表单控件包括输入文本框（Input）、文本区域（Textarea）、下拉列表（Select）等。使用这些不同类型的控件的目的都是为了信息输入，它们会有一些公共的属性，所以我们把表单控件抽象出一个基类并统一起来。统一表单控件有很多好处，一方面可以确保代码结构的一致，有利于维护和扩展；另一方面也有利于保持应用的风格，在多人合作开发的情况下还能提升开发效率。同时，表单也要考虑到校验问题，这是上面的例子没有涉及的。下面将主要介绍表单结构及表单校验等内容。

17.2 表单控件组件

17.2.1 定义表单控件

先来看单个表单控件组件的目录结构：

```
field
├── field-base.ts
├── field-text.ts
├── field-validators.ts
├── field.component.css
├── field.component.html
├── field.component.ts
└── index.ts
```

从目录结构上来讲，field-base.ts 是表单控件的基础，在这里定义了表单控件应该包含的属性。示例代码如下：

```
// user/shared/field/field-base.ts

export class FieldBase<T>{
  value: T;
  key: string;
  label: string;
  required: boolean;
  pattern: string;
  order: number;
  controlType: string;
  constructor(options) {
    // ...
  }
}
```

FieldBase 基础类中各属性说明如表 17-1 所示。

表 17-1 表单基础类的属性说明

属性名称	含义	类型
value	控件的值	泛型
key	名称，用于唯一标识一个表单中的某个表单控件	字符串
label	控件显示的名称，用于展示，与 key 对应	字符串

续表

属性名称	含义	类型
required	是否是必填项	布尔值
pattern	校验规则	字符串
order	顺序	数值
controlType	控件类型	字符串

有了这个基础类之后，我们就可以进行下一步的扩展变化了。以最常见的 <input> 标签为例，新建 FieldText 类并继承 FieldBase 基础类，且扩展一个新的 type 属性，因为 <input> 标签的 type 属性包含了例如 text、password、checkbox 等类型值，不同 type 属性的控件在 UI 展示或者功能上是有一定的差别的。FieldText 控件的示例代码如下：

```
// user/shared/field/field-text.ts

import { FieldBase } from './field-base';
export class FieldText extends FieldBase<string> {
  controlType = 'text';
  type: string;
  constructor(options: any) {
    super(options);
    this.type = options['type']; // 设置 input 类型
  }
}
```

在问卷调查系统中还包含 FieldRadio 控件（field-radio.ts）和 FieldSelect 控件（field-select.ts），它们的实现方式与 FieldText 控件类似，这里就不再重复说明了。

17.2.2 校验表单控件

在实际开发中，表单提交功能或多或少都会涉及输入数据的校验。校验包括前端校验和后端校验，本章所说的的校验都是指前端校验。前端校验的好处很明显，一方面可以提升用户体验，让用户第一时间知道该输入什么，如何对不合法的输入进行修改；另一方面可以减少无效的提交，减少和服务端的交互，从而提升提交的成功率。一般来说，浏览器支持部分原生校验，比如 required、maxlength 等。但是，浏览器原生提供的校验一般都不能满足全部需求，为此，我们还需要自定义一些校验规则，以满足特定的需求，并且给出更友好的错误提示。前面提到的 FieldBase 基础类中的校验规则 pattern 就是用来指定校验规则的，自定义校验规则的示例代码如下：

```
// user/shared/field/field-validators.ts

import { FormControl } from '@angular/forms';

// 定义用于内容校验的正则表达式
const REG = {
  USERNAME: /^\\w{1,20}$/,
  PASSWORD: /^\\w{6,20}$/
};

// 定义校验结果的数据结构
interface ValidationResult {
  [key: string]: boolean;
}

export class FieldValidators {

  // 定义 username 校验规则
  static username(control: FormControl): ValidationResult {
    if (control.value.length === 0) {
      return {
        empty: true
      };
    }
    if (REG.USERNAME.test(control.value)) {
      return null;
    }
    return {
      invalid: true
    };
  }
  // ...
}
```

接下来看在表单控件的组件模板中有关校验提示的部分。示例代码如下：

```
<!-- user/shared/field/field.component.html -->

<div [formGroup]="form">
  <label [attr.for]="field.key">{{field.label}}</label>
  <div [ngSwitch]="field.controlType">
```



```

    <input *ngSwitchCase="'text'" [formControlName]="field.key"
      [id]="field.key" [type]="field.type">
  </div>
  <!-- ... -->
  <div class="errorMessage" *ngIf="!isEmpty && !isValid">{{field.label}}格式不正确
    </div>
  <div class="tipsMessage" *ngIf="isEmpty">请填写{{field.label}}</div>
</div>

```

最后看在 FieldComponent 组件中是如何实现检验提示的判断逻辑的。示例代码如下：

```

// user/shared/field/field.component.ts

// ...

@Component({
  selector: 'field',
  templateUrl: 'field.component.html',
  styleUrls: ['field.component.css']
})
export class FieldComponent {

  @Input() field: FieldBase<any>;
  @Input() form: FormGroup;

  // 通过 isValid() 判断是否校验通过
  get isValid() {
    return this.form.controls[this.field.key].valid;
  }

  get isEmpty() {
    // 注意这里的 errors 可能为 null，对于 null 不能直接取 empty
    let errors = this.form.controls[this.field.key].errors || {};
    return errors['empty'];
  }
}

```

这里需要特别说明的是，在组件模板中将会根据校验结果呈现以下三种检验状态：

- 当输入为空的时候，显示输入提示。

- 当输入不为空且输入校验失败时，显示错误提示。
- 当输入检验成功时，显示正确提示或不显示任何提示。

举个例子，当用户有输入操作时，将在自定义校验类 `FieldValidators` 中根据 `control.value.length` 的值判断输入是否为空，最终在模板中根据 `isEmpty` 访问器函数返回的校验结果来判断是否显示错误信息。

以上便是表单组件及校验的逻辑。如果需要扩展一个新的控件，比如 `Checkbox`，只需要定义一个新的类 `FieldCheckbox`，并在 `field.component.html` 模板文件中根据模板语法编写模板即可。如果需要新增一个校验规则，只需要在 `field.validators.ts` 自定义校验类中扩展一个新方法即可。其实完整的校验功能还应该支持异步的 `Validator`、支持动态配置的错误或提示语模板等，受篇幅所限这里就不再展开介绍了。

17.2.3 表单安全

除表单控件验证外，表单安全也是 Web 开发者应该重点关注的领域，特别是涉及登录、注册、支付这类敏感操作时。常见的攻击包括跨站脚本（XSS）、跨站请求伪造（XSRF）、跨站脚本包含（XSSI）等。

跨站脚本允许攻击者将恶意代码注入到页面中，并可能窃取登录数据等安全信息，同时还可以冒充用户执行操作。为此 `Angular` 针对一些可能导致安全隐患的场景，包括 `HTML` 属性绑定，例如 `innerHTML`、`style` 样式绑定、`URL` 绑定等提供了安全环境（`Security Context`）保护，并对里面的内容进行无害化处理（`Sanitization`），主要是过滤掉 `<script>` 标签等不可信任的值，并转化成可以安全插入到 `DOM` 中的数据格式。

另一方面，有时候开发者确实需要把 `HTML`，甚至包括 `<script>` 标签内容动态渲染到页面上，比如富文本编辑这种场景，此时可以调用 `DomSanitizer` 服务的 `bypassSecurityTrustScript()`、`bypassSecurityTrustUrl()`、`bypassSecurityTrustResourceUrl()` 等方法跳过 `Angular` 的默认安全化处理，建议开发者充分熟悉应用场景并慎重评估后再跳过安全化处理。

一般而言，在开发过程中应该避免直接使用原生 `DOM API` 操作页面内容、动态生成模板替换页面，或者选用未支持无害化处理的服务模板引擎等。因为这些都会绕过 `Angular` 的安全处理机制，同时也不利于 `Angular` 代码的跨平台使用，如以后选用服务端渲染 `Universal` 等技术，所以尽量慎用。

了解完部分 `Angular` 表单的安全处理后，下面将以表单控件为基础，创建问卷调查系统的用户注册组件。

17.3 用户注册功能开发

如何在系统里面唯一标识一个用户，最简单常用的方法就是让用户先注册，并提交必要的身份信息。上一节已经完成了表单控件的开发，接下来将讲解如何基于表单控件完成用户注册组件的开发。有了上述表单控件的基础，要新建一个表单就简单多了，开发者可以不用关注控件的具体表现和逻辑，只需要根据业务情况定义相关的表单控件就可以了。首先来看注册组件（RegisterComponent）的代码目录结构：

```
|—— index.ts  
|—— register.component.css  
|—— register.component.html  
|—— register.component.ts
```

用户注册组件相关文件作为一个整体放在 user/shared/register 目录下，注册相关服务则放在公共的 core 目录下。

17.3.1 用户注册服务

为了便于开发和维护，我们通常会抽象出一个负责数据交互和处理业务逻辑的服务。

在这里定义一个用户对象，并指定注册时要填写的信息，本例中只定义了用户名 username 和密码 password。需要指出的是，在不同系统里要求填写的信息可能不一样，考虑到找回密码的情况，一般还会要求填写邮箱等信息，但不管怎样，用户名和密码都是必不可少的（使用第三方联合登录的情况除外）。为了简化问题，这里没有加入邮箱等信息的填写，因为完整的邮箱注册还包括发送邮件验证邮箱正确性等步骤，而这些步骤跟本书要讲的内容并没有直接关系。注册服务的示例代码如下：

```
// core/services/register.service.ts  
  
// ...  
import { FieldBase } from '../../user/shared/field/field-base';  
import { FieldText } from '../../user/shared/field/field-text';  
import { FieldValidators } from '../../user/shared/field/field-validators';  
import { SITE_HOST_URL } from '../../shared/index';  
  
@Injectable()  
export class RegisterService {  
  
    private registerUrl = `${SITE_HOST_URL}user/add`;
```

```

constructor(private http: Http) { }

getFields() {
    // ...
}

toFormGroup(fields: FieldBase<any>[]) {
    // ...
}

addUser(data: Object) {
    // ...
}
}

```

在 RegisterService 服务中，对外提供了 `getFields()`、`toFormGroup()`、`addUser()` 三个方法，这些方法的功能具体说明如下。

- `getFields()`: 定义了用户注册涉及的用户名和密码属性，并指定了其默认值和校验规则。
- `toFormGroup()`: 将表单控件和表单关联起来并设置相应的校验规则。
- `addUser()`: 与后台接口进行数据交互，用于提交用户注册信息。

需要指出的是，当需要同时应用多个校验规则时，只需要在创建 `FormControl` 时使用 `Validators.compose([...pattern])` 将多个规则作为参数传入即可。

17.3.2 组件的逻辑

现在终于到了实现注册逻辑的时候了。有了上面的通用表单控件，组件逻辑的编写就简单多了，只需要定义用户名和密码两个表单控件即可。当然，在实际项目中可能会加入验证码等功能，这里简化不做相关的实现。通过用户注册组件的模板代码可以看出用户注册功能的整体结构。示例代码如下：

```

<!-- user/shared/register/register.component.html -->

<div>
  <div *ngIf="!registered">
    <h2>注册</h2>
    <form (ngSubmit)="register()" [formGroup]="form" class="form-group form-group-

```

```

    lg">
    <div *ngFor="let field of fields" class="form-row">
      <field [field]="field" [form]="form"></field>
    </div>
    <div class="form-row form-checkbox">
      <input type="checkbox" id="showPassword" (click)="showPassword()">
      <label for="showPassword">显示密码</label>
    </div>
    <div class="form-row text-center">
      <button type="submit" class="btn btn-default btn-lg" [disabled]="!form.
        valid">提交</button>
      <button type="reset" class="btn btn-default btn-lg" (click)="resetForm()">
        重置</button>
    </div>
  </form>
</div>
<div *ngIf="registered">
  <alert [type]="alert.type" dismissible="true" >
    {{ alert?.msg }}
  </alert>
</div>
</div>

```

接下来看用户注册组件（RegisterComponent）的表单交互逻辑部分。示例代码如下：

```

// user/shared/register/register.component.ts

// ..
import { FieldBase } from '../field/field-base';
import { RegisterService } from '../../core/services/register.service';
import { UserService } from '../../core/services/user.services';

@Component({
  selector: 'register',
  templateUrl: 'register.component.html',
  styleUrls: ['register.component.css']
})
export class RegisterComponent implements OnInit {

```

```
form: FormGroup;
registered = false;
fields: FieldBase<any>[] = [];
alert:any = {msg: '注册成功', type: 'success', closable: true};

constructor(private rs: RegisterService,
              private userService:UserService,
              private route:Router) {
  this.fields = rs.getFields();
}

ngOnInit() {
  this.form = this.rs.toFormGroup(this.fields);
}

// 显示密码明文的逻辑
showPassword () {
  this.fields.forEach((field : any) => {
    if (field.key === 'password') {
      field.type = field.type === 'password' ? 'text' : 'password';
    }
  });
}

// 表单重置
resetForm () {
  this.form.reset();
}

// 用户注册
register() {
  // ...
}
}
```

用户注册组件初始化时，首先在构造方法中调用 `getFields()` 方法，完成表单控件的初始化，再在 `ngOnInit()` 方法中进行表单初始化，在这里直接调用了上一节中 `RegisterService` 服务提供的 `toFormGroup()` 方法，它背后实际上执行的是 `new FormGroup()`。

另外，用户注册组件里还有 `showPassword()`、`resetForm()` 和 `register()` 三个方法，它们对应的交互逻辑具体说明如下。

- 当用户选中“显示密码”复选框时，调用 `showPassword()` 方法修改密码控件的 `<input>` 标签的 `type` 属性为 `text`。
- 当用户单击“重置”按钮时，调用 `resetForm()` 方法重置表单，还原表单为默认状态。
- 当用户单击“提交”按钮时，调用 `register()` 方法进行表单提交，并处理返回结果。

17.3.3 注册接口开发

上面主要介绍了前端部分的开发，与之对应的后端注册接口为 `/api/user/add`。示例代码如下：

```
// backend/app.js

let crypto = require('crypto');

const md5 = str => crypto
  .createHash('md5')
  .update(str.toString())
  .digest('hex');

// ...
server.post('/user/add', (req, res) => {
  let item = req.body;
  // 查找是否有相同的用户名
  let user = db('user').find({
    username: item.username
  });
  if (user) {
    // 如果存在相同的用户名，则提示用户已存在
    res.json({
      success: false,
      message: '"' + item.username + '" is exists'
    });
  } else {
    // 如果不存在相同的用户名，则保持用户信息并完成注册
```

```
// 这里对密码做了 MD5 加密存储
item.password = md5(item.password);
// 同时保存注册时间
item.createDate = new Date().toLocaleDateString();
db('user').push(item).then(() => {
  res.json({ success: true });
});
}
});
// ...
```

后端注册接口的实现比较简单，首先判断用户名是否存在，然后对密码做 MD5 处理后再做存储。

17.4 权限管理

认证（Authentication）与授权（Authorization）是权限管理模块的重要组成部分。

认证用于校验用户身份，授权则检查认证后的用户是否有特定行为的操作权限。在问卷调查系统中，创建问卷、问卷编辑和我的问卷这些功能是需要登录并且有相应权限才能进入的。为了简化处理，在本示例中只要用户登录验证通过了，其就有创建、编辑或查看我的问卷的权限，而没有完成登录认证的用户都会被强制跳转到登录页面，登录流程如图 17-3 所示。

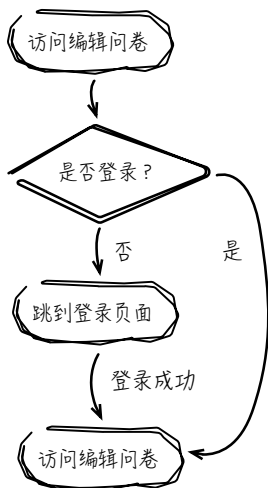


图 17-3 登录流程图

路由拦截

在第 11 章路由章节中提到过，Angular 提供了多种类型的路由拦截特性，这些特性允许在跳到一个页面之前执行指定的逻辑，并根据执行的结果来决定是否进行跳转。其中 CanActivate 接口很适合做权限控制，可以最终通过一个布尔值来决定是否激活目标路由。因此，利用 Angular 路由提供的特性，实现 CanActivate 接口并根据业务场景添加到相应的路由配置中，即可实现通用的页面鉴权功能。自定义的 AuthGuard 示例代码如下：

```
// core/services/auth-guard.service.ts

import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot } from '@angular/router';

import { UserService } from '../user.service';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private userService: UserService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    if (this.userService.isLogin) {
      return true;
    }
    this.router.navigate(['/admin/login'], {queryParams: {returnUrl: state.url}});
    return false;
  }
}
```

在上述代码中，用 this.userService.isLogin 来判断用户是否已登录，isLogin 这个属性在项目中被赋值为 true 的地方通常有三处，一是系统启动时通过调用 user 服务的 getUser() 方法成功获取用户信息后；二是在用户登录成功后；三是在注册成功后。需要注意的是，getUser() 方法背后的 HTTP 请求通常要靠 Cookie 里的类似于 SessionID 的标识到后台换取登录态及用户信息，但本项目只专注前端部分，后端并没有加上 Cookie 相关处理，而在实际项目中运用 Cookie 是不可或缺的技术。

如果用户未登录，则会跳到登录页面，注意后面的参数 queryParams: {returnUrl: state.url}。在登录的场景中，通常用户希望在登录成功后自动跳转回登录前的页面，这种体

验比较好，在本项目中也是这么实现的。`returnUrl` 参数把当前的 URL（通常还需要做转义处理）传递到登录页面中，在登录页面成功认证后根据该参数跳转回原来的页面。登录的相关逻辑示例代码如下：

```
// user/shared/login/login.component.ts

// ...
export class LoginComponent implements OnInit {
  // ...
  login() {
    this.loginService
      .login(this.form.value)
      .subscribe((res: Response) => {
        let body = res.json();
        if (body && body.success) {
          this.userService.isLogin = true; // 设置用户已登录
          this.userService.userInfo = { username: this.form.value.username,
            createDate: new Date().toLocaleString() }
          this.route.navigateByUrl(this.returnUrl?this.returnUrl:'/'); // 如果有传
            递returnUrl，跳转回目标页，否则去到首页
        }
      }, error => {
        console.error(error);
      });
  }
}
```

至此，用户管理模块的内部业务逻辑处理基本完成，最后只需对需要鉴权的页面添加路由拦截配置即可，例如前面章节中提到的后台管理模块。实际上需要用户登录之后才能进入后台管理，所以需要对后台管理模块接入鉴权功能。示例代码如下：

```
// admin/admin-routing.module.ts

const routes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard], // 增加 canActivate 控制路由进入
    children: [
      {
```

```
    path: '',
    children: [
      { path: 'center', component: CenterComponent },
      { path: 'edit/:id', component: EditComponent }
    ]
  }
]
};
```

由上面的代码可以看出，路由配置只需要增加 `canActivate: [AuthGuard]` 这行配置项，即可完成鉴权控制。它实现起来很方便，并且这种配置也是通用的，可直接在任意路由做插拔管理。

最终，登录页面的展示效果如图 17-4 所示。

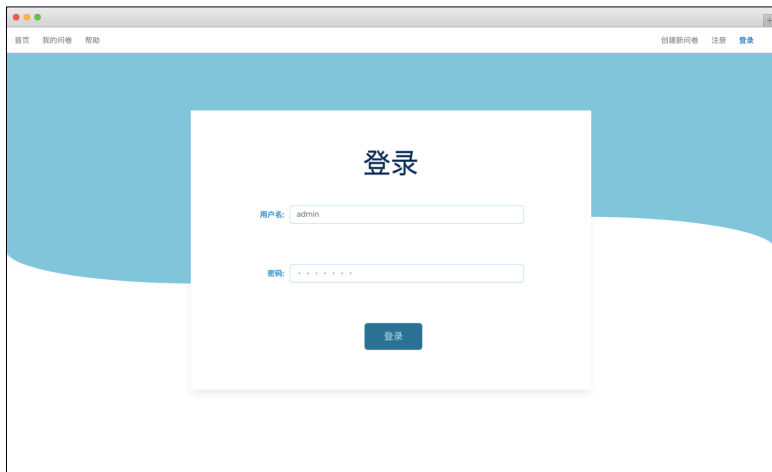


图 17-4 用户登录页面的展示效果

17.5 小结

本章先对注册功能的实现进行了讲解，主要包括常见的表单处理、将表单控件抽象成组件、表单验证及表单安全等知识点。在组件化开发的应用中，一个应用通常是由一个个组件组合出来的，考虑到项目的扩展性、可维护性和代码复用性，将常用的功能逻辑抽象成组件（比如示例中的表单控件组件）是很有必要的。另外，登录组件和注册组件基本一致，都是基于表单控件组件开发的，所以本章简化了关于登录组件的相关介

绍。最后通过认证授权这个场景介绍了 Angular 的路由拦截特性，利用这个特性可以实现通用的登录认证和权限鉴别等操作。

至此，我们就介绍完了整个问卷调查系统的功能。下一章将介绍项目自动化构建相关内容，以及使用 Angular 的一些最佳实践，用以帮助开发者提高开发效率，编写出高质量的项目代码。

18

项目构建和最佳实践

前面几章的内容主要介绍了问卷调查系统的整个实现过程，通过实际项目的开发实践来使读者加深对 Angular 相关技术知识的理解。但在整个项目生命周期中，还需要完成诸如代码质量检查、测试、压缩、打包等构建工作，只有一步步实施完这些步骤，我们的项目才能在代码可维护性、产品功能健壮性、资源分发有效性上得到保障。

本章首先讲解问卷调查项目中的代码质量检查、测试、打包等关键构建步骤，然后进一步讲解开发 Angular 应用应该遵循的最佳实践。

18.1 项目构建

Angular CLI 不仅提供了 build 命令实现项目的打包构建，还提供了 lint 命令完成对代码标准的检测，以及 test 命令执行项目的测试工作。

18.1.1 代码质量检查

代码质量检查（Linting）是分析并检查源代码中可能出现错误（包含静态分析和代码风格）的过程，它在一定程度上保证了代码质量，提升了代码的可读性和可维护性。

TSLint 是 TypeScript 语言下的可扩展代码静态分析工具，lint 命令使用 TSLint 定义的规则分析代码，并给出修改建议，保证了代码的质量和风格。

lint 命令支持多个参数，例如指定 fix 参数可以自动修复检测到的代码错误，命令如下：

```
$ ng lint --fix=true
```

除此之外，还有 force 参数，它标识是否强制检测通过；type-check 参数标识是否进行类型检测；format 参数定义检测结果的输出格式等。

开发者应该把代码质量检查加入到项目构建流程中，确保所提交的代码是经过静态检查和代码风格检查的，以保证代码质量和风格统一。

18.1.2 测试

开发者通过编写并运行测试用例来确保应用的正常运行，防止代码在日常迭代或重构中引入新的问题，同时也保证了代码运行能达到预期的效果。在实现具体的测试例子之前，首先了解 Angular CLI 提供的 test 命令。

执行 test 命令，首先会通过 Karma 构建应用程序。Karma 是基于 Node.js 实现的 JavaScript 测试执行过程管理工具，它可以监控文件的变化，然后自动执行测试用例。不过在执行 test 命令时，开发者也可以通过指定参数 --watch=false 或者 --single-run 来确保命令只会被执行一次。

除此之外，参数 --code-coverage 可用来生成测试覆盖率报告。报告文件将被存储到 coverage 文件夹中。

具体的测试技术在本书第 12 章测试章节中已经有较为系统的介绍，下面只展示如何在问卷管理项目中使用它。

接下来以帮助页面为例，对 AboutComponent 组件进行单元测试，在它同级目录下新建同名且带有 .spec 后缀的文件，示例代码如下：

```
// about.component.spec.ts

export function main() {
  describe('About component', () => {

    let comp: AboutComponent;
    let fixture: ComponentFixture<AboutComponent>;
    let aboutEl: DebugElement;

    beforeEach( async(() => {
      TestBed.configureTestingModule({
```

```

    imports: [SharedModule, AccordionModule], // 导入要测试的模块
    declarations: [AboutComponent], // 声明被测试的组件
  })
  .compileComponents(); // 编译模板跟 CSS
}));

// synchronous beforeEach
beforeEach(() => {
  fixture = TestBed.createComponent(AboutComponent);
  comp    = fixture.componentInstance;
  aboutEl  = fixture.debugElement;

  fixture.detectChanges(); // trigger initial data binding
});

it('should render accordion', ()=>{
  const de = aboutEl.queryAll(By.css('accordion'));
  expect(de.length).toBe(1);
});

it('should render correct accordion text', ()=>{
  // const de = aboutEl.queryAll(By.css('accordion'));
  expect(aboutEl.nativeElement.textContent).toContain('常见FAQ');
});
});
}

```

运行 `ng test` 命令来执行单元测试，测试结果如下：

```

// ...
START:
09 12 2016 11:54:22.705:INFO [karma]: Karma v1.3.0 server started at http://
  localhost:9876/
09 12 2016 11:54:22.708:INFO [launcher]: Launching browser Chrome with unlimited
  concurrency
09 12 2016 11:54:22.715:INFO [launcher]: Starting browser Chrome
09 12 2016 11:54:25.703:INFO [Chrome 54.0.2840 (Mac OS X 10.11.6)]: Connected on
  socket /#yGj76-oEv0xTvyKfAAAA with id 84290634
About component
  ✓ should render accordion
  ✓ should render correct accordion text

```

Finished in 0.75 secs / 0.712 secs

SUMMARY:

✓ 2 tests completed

18.1.3 打包

代码质量检查和测试回归通过后，我们需要构建并发布应用。熟悉前端工程化的读者都知道，项目文件的构建打包一般包含脚本文件预处理、图片 Sprite 化、资源合并压缩、文件指纹戳生成、文件内容的引入替换及 CDN 化等。在第 14 章中介绍了 build 命令的使用，本节将继续深入讲解 Angular 的打包构建方法。

在之前章节中介绍到启动 Angular 应用有两种不同的引导方式，即动态引导和静态引导。

Angular 应用程序主要由组件及其 HTML 模板组成。在运行前，组件和模板必须由 Angular 编译器转换为可执行的 JavaScript 程序。采用动态引导加载时使用的是 Just-in-Time (JiT) 编译方式，即在浏览器中编译 Angular 应用，但这样会导致运行时的性能损耗，增加了首屏渲染时间。

而静态引导使用的是 Ahead-of-Time (AoT) 编译方式，即在构建时预先编译应用代码，而不通过浏览器编译，从而使得应用程序的启动速度更快，并且 AoT 可以提前捕获模板在数据绑定中出现的错误，提升了代码质量。

这两种引导方式的不同点表现在：在启动应用时，JiT 是通过 `platformBrowserDynamic.bootstrap()` 方法引导 `AppModule` 的；而 AoT 则通过 `platformBrowser.bootstrapModuleFactory()` 方法引导生成好的模块工厂 `AppModuleNgFactory` (`AppModule` 编译后生成的文件)。

在项目中，建议使用 AoT 方式预先编译应用，执行如下命令即可：

```
$ ng build --aot
```

除此之外，在构建的过程中还通过 Tree Shaking 技术做了更进一步的优化，它从上到下遍历依赖关系，就像摇动树木使枯死的树枝树叶落下一样，移除那些未使用的框架代码，最终打包的文件只包含应用使用到的代码块，在一定程度上减少了应用程序包的大小。比如简单的应用可能不会用到 `@angular/routes` 路由功能，所以 Tree Shaking 就可以把路由部分代码从最终打包好的文件中移除。如果还想要做更极致的优化，则可以引入 Rollup 来处理，感兴趣的读者可以自行去学习，这里不再赘述。

18.1.4 容器化

除上面提到的在本机按照依赖来编译和构建的方式外，我们还可以结合 Docker 容器化技术来构建开发和生产的运行环境，为 Angular 应用的运行环境提供容器化选择，甚至可以结合 Gitlab CI/Jenkins 来完成持续化集成构建，当有新代码合入代码仓库的 master 或者 release 分支时，就触发代码质量检查、测试及打包等构建工作，并以 Docker 镜像的方式提供最终代码。在开发模式下，此容器提供编译后的应用程序文件和服务；而在生产模式下，它构建被压缩后的应用程序文件，最终由 CDN 或 Nginx 之类的容器对外提供服务。

可以使用如下 docker-compose 命令来运行测试环境，运行成功后在浏览器中打开 `http://localhost:5555` 即可看到页面。

```
$ docker-compose build
$ docker-compose up -d
```

在生产环境下构建和部署，则运行如下命令：

```
$ docker-compose -f docker-compose.production.yml build
$ docker-compose -f docker-compose.production.yml up angular-seed # 等待该容器完成构建，因为 Nginx 依赖它生成的构建物
$ docker-compose -f docker-compose.production.yml up -d angular-seed-nginx # 在后台 detach 模式下启动 Nginx 服务器
```



docker-compose 命令需要安装 Docker 环境才能使用。Docker 是一种轻量的虚拟化容器技术，我们可以使用 Docker 技术来方便地构建标准化的运行环境。关于 Docker 的知识已经超出本书的范围，读者可以自行了解。

18.2 最佳实践

在本部分的实战项目中，细心的读者会发现，无论是代码文件的目录结构、代码组织，还是文件或类的命名，我们都遵循了一定的规范，这使得项目代码看上去很整洁，也便于查找，可读性和可维护性非常好。所以良好的编码习惯和编码规范是写出让人赏心悦目的代码的前提。

在本节中，我们根据实际项目的开发经验并结合 Angular 官方提供的风格指南，总结了一些 Angular 技术开发的最佳实践，并尝试讲解了应该遵循这些最佳实践的原因。

18.2.1 单一职责

Angular 是基于 Web Component 思想设计的，所有的核心概念包括组件、服务、指令等都遵循单一职责原则，这是最佳实践中很重要的一条，它有助于代码保持良好的可读性和可维护性，而且极大地方便了单元测试。在 Angular 应用中创建的所有组件、服务或其他模块都应该遵循单一职责原则，以下是关于单一职责原则在 Angular 开发中的具体要求。

单一文件

单一规则是指在一个文件中，应该只定义一个组件、服务或指令等。

一个组件对应一个文件可以保证代码简单、易读，具有更好的可维护性。通常一个文件的代码量不应超过 400 行，如果太长，则可能是因为组件拆分得不够细致。而且在团队开发中，不同成员修改的功能存在于不同的文件中，可以在一定程度上避免跟其他成员的修改产生冲突，大大减少了耦合。例如，在问卷调查系统的组件树中，首页组件 HomeComponent、登录组件 LoginComponent、注册组件 RegisterComponent 等一一对应着 home.component.ts、login.component.ts、register.component.ts 等文件，符合一组件一文件的规则。

将多个组件写在同一个文件中往往会不可避免地产生变量共享或耦合，或者为了规避相互影响而采用闭包，但这样往往会导致另外一些隐藏的问题。而单一的组件可以作为一个文件的默认导出对象，在使用路由进行懒加载时，可以避免加载到不需要的组件代码，其带来的好处不言而喻。

简单函数

简单函数是指所定义的函数尽量功能简单、目的明确、职责单一。

保持函数简单的好处和单一规则类似，都是为了代码的易读、易维护。通常一个函数的代码量不应超过 75 行，目的更明确且职责更单一的函数具有更高的可复用性，也便于进行测试。

18.2.2 命名约定

关于命名约定是一个老生常谈的问题，但它确实是编写高可维护性和易读性代码的非常重要的规范。良好的命名规范能起到见名知意的效果，能使维护代码的开发者在只通览代码文件结构或名称的情况下就能大致了解整个项目，并且在需要了解细节的情况下能快速定位到相应的代码位置。对于 Angular 项目，也有关于命名约定方面的最佳实践。

命名风格统一

使用统一的命名风格是指保持命名方式和命名模式一致。在 Angular 中通常采用 `feature.type.ts` 的命名模式，即先描述特性，再描述类型。例如用户列表组件的文件名为 `user-list.component.ts`、用来做日志上报的日志服务文件名为 `logger.service.ts`、自定义管道文件名为 `ellipsis.pipe.ts` 等。

统一命名风格的好处很明显，例如我们可以通过在 IDE 中搜索 `home` 关键词就能找出所有与首页相关的组件、服务、模板等文件，或者通过搜索 `component` 关键词找出项目中定义的所有组件文件等。并且通过这些文件的名称，开发者可以轻易地判断出这个组件是为哪个功能特性编写的。这极大地提高了定位代码的效率，对于越大型的项目其作用越明显。

名称格式保持一致

对于文件的名称，通常使用 “.” 点号来分隔特性和类型，而对于特性名称由多个英文单词组成的情况，则可以使用 “-” 短横线作为分隔符。例如上面提到的用户列表组件的文件就可以命名为 `user-list.component.ts`。

使用完整的类型名称，不要使用简写形式。例如使用 `.service` 作为服务的类型描述，而不要使用 `.srv`、`.svc`、`.serv` 等容易让人产生困惑的名称。

使用一致的名称格式还有一个好处，就是可以方便地使用正则表达式匹配的方式做一些自动化任务。例如自动在组件文件开头添加统一的文件注释、版本号等。

选择器命名约定

对于组件的选择器，通常采用 “烤肉串” 命名形式。例如问卷调查系统实例中的问题大纲组件，其 `selector` 属性的值为 `questionnaire-outline`，示例代码如下：

```
// ...
@Component({
  inputs: ['questionnaire'],
  selector: 'questionnaire-outline',
  template: '...'
})
export class QuestionnaireOutline {
  private questionnaire: QuestionnaireModule;
  constructor() {}
}
```

而对于一般指令的选择器，通常采用首字母小写的驼峰命名形式，以便与组件的选择器命名区分开。例如在本书第二部分中讲解的自定义指令 `unless`，示例代码如下：

```
@Directive({selector: '[myUnless]'})
export class UnlessDirective {
  @Input('myUnless') condition: boolean;
}
```

测试相关文件的命名约定

测试文件包括单元测试文件和端到端测试文件。一般而言，单元测试文件的命名应该与被测试的组件或者服务的文件名称保持一致，并添加 `.spec` 后缀。例如 `HomeComponent` 组件的测试文件命名为 `home.component.spec.ts`，`LoggerService` 日志服务的测试文件则命名为 `logger.service.spec.ts`。而端到端测试文件名称的后缀应为 `.e2e-spec`，以便与单元测试文件区分开，如 `app.e2e-spec.ts`。

命名的前后缀

有时候我们应该为文件的命名添加一些前缀或后缀。假如在用户管理系统中，普通用户看到的用户成员列表和管理员看到的成员列表样式功能可能有所不同，需要使用两个组件分别实现，这时候就需要给组件定义它的使用范围，并通过命名前缀来区分。例如普通用户的成员列表组件命名为 `users.component.ts`，而管理员的成员列表组件命名为 `admin-users.component.ts`。

在必要的时候，也可以给自定义的指令添加前缀。例如定义一个表单校验指令 `validate`，由于它是一个通用组件，可以被其他模块或者项目复用，名称也很常见，因此最好添加一个有意义的前缀，以便与其他表单验证指令区分开。例如可以以公司简称（`gf`）等作为前缀来命名，如通用表单校验指令命名为 `gfValidate`，示例代码如下：

```
// ...
@Directive({
  selector: '[gfValidate]'
})
export class ValidateDirective {}
```

由上可见，添加前缀的好处是一方面可以避免命名的冲突；另一方面可以界定组件的作用范围。

启动文件的命名约定

启动文件是 Angular 中很重要的一个文件,是程序的入口文件。通常都是使用 `main.ts` 作为文件名称的,让代码的读者一眼便能找到程序入口。启动文件的内容应该只包含和平台相关的启动代码,不应包含跟应用逻辑相关的任何代码,逻辑代码应该放到组件或服务中。

18.2.3 编码约定

编码约定是指在具体代码中一些约定俗成的命名或格式,这些编码约定如下。

类

类名采用首字母大写的驼峰命名形式。不论是组件类名、服务类名,还是指令类名,命名一律遵循首字母大写的驼峰命名约定。请注意不要与文件的命名或选择器的命名混淆,例如问卷编辑组件中的相关命名,示例代码如下:

```
// edit.component.ts -> 文件名称
import { Component } from '@angular/core';

@Component({
  selector: 'edit', // -> 选择器名称
  styleUrls: [ 'edit.component.css' ],
  templateUrl: 'edit.component.html'
})
export class EditComponent{ // -> 类名称
  constructor(){}
}
```

常量

在 TypeScript 中,对于在整个应用的生命周期中都不会改变的变量,则可以使用 `const` 关键字修饰为常量。常量命名采用全字母大写并用下划线分隔单词的形式,形如 `UPPER_SNAKE_CASE`,示例代码如下:

```
export const ARR_CODE = ['000300', '000001']; // 常量数组
export const URL = 'angular.io'; // 常量字符串
```

接口

接口名的定义方式与普通类的定义方式一致，即首字母大写的驼峰命名形式。在传统面向对象语言中会在接口名前添加大写字母 I，但是在 TypeScript 语言中并不建议这么做。例如在问卷数据模型中对问卷数据接口的定义方式如下：

```
export interface QuestionnaireModel {  
  id?:number; // 问卷ID  
  title:string; // 问卷标题  
  owner:number; // 作者ID  
  starter:string; // 开始问候语  
  ending:string; // 结束问候语  
  state:QuestionnaireState; // 问卷状态  
  questionList: QuestionModel[]; // 问题列表  
  createDate?:string; // 创建日期  
}
```

属性和方法

属性和方法使用小写字母开头的驼峰命名形式即可。不要画蛇添足地给私有属性或方法添加“_”下画线前缀。因为对于 JavaScript 而言，它没有真正的私有属性或方法，而在 TypeScript 中又有 `private` 关键字可以标识私有成员，因此并不需要下画线来特别强调。一般来说，属性定义应该放在方法定义之前，而公有成员定义应放在私有成员定义之前。属性和方法相关规范的示例代码如下：

```
export class DataService {  
  
  message: string; // 公有成员放在私有成员前  
  private tempData: number;  
  
  getData() {  
    this.tempData++;  
    this.log();  
    return message;  
  }  
  
  private log() { // 私有成员不需要添加下画线前缀  
    console.log(this.message);  
  }  
}
```

关于 import 语句

当使用 import 语句导入一组对象时，在括号内侧加上一个空格，便于阅读，示例代码如下：

```
// 代码显得过于紧凑，不推荐
import {HttpClient, HttpHeaders} from '@angular/common/http'; // 代码显得过于紧凑
import {Injectable} from '@angular/core';
```

```
// 带上空格看起来更清晰，推荐
import { HttpClient, HttpHeaders } from '@angular/common/http'; // 带上空格看起来更清晰
import { Injectable } from '@angular/core';
```

当使用 import 语句导入对象时，导入对象最好按字母顺序排序。另外，我们约定将导入的第三方库和项目本身的代码用空行隔开，示例代码如下：

```
import { Component, Input, OnInit } from '@angular/core';
import { FormGroup, REACTIVE_FORM_DIRECTIVES } from '@angular/forms';

// 使用空行将导入的 Angular 自带的库文件隔开
import { FieldBase } from '../field/field-base';
import { FieldComponent } from '../field/field.component';
import { RegisterService } from './register.service';
```

应用结构约定

上文中所提及的最佳实践都是立足于单个文件或单个类的，关于应用结构方面的约定则是站在整个应用的角度来考虑的。任何项目都是从很小的项目开始的，但是随着功能的不断迭代，代码会变得越来越庞大，如果没有对代码结构进行良好的组织，项目很快就会变得难以维护，甚至到了最后不得不进行重构的地步。因此，在项目初始阶段就规划好应用的整体结构是非常有必要的，下面列举的一些约定有助于帮助开发者规划代码结构。

- 所有的项目代码应该存放在一个名为 `app` 的文件夹中。
- 将第三方代码存放在 `app` 目录之外的独立文件夹中。
- 以功能特性创建并命名文件夹。例如在问卷调查系统中，首页、登录、注册、创建问卷、编辑问卷等功能都存放在相应独立的文件夹中，这些文件夹的名称都跟功能模块的特性有所关联。

- 将一个模块内的所有共享文件放在一个名为 `shared` 的文件夹中。例如在问卷创建模块中，关于问卷数据模型（问卷模型、问题项模型及问卷状态等）的定义可以集中存放在问卷创建模块文件夹下的 `shared` 文件夹中。
- 将用于全局布局的组件放在 `shared` 目录中，如导航（`nav`）组件、页脚（`footer`）组件等。
- 使用 `index.ts` 文件作为一系列相关模块的统一导出文件。例如在存放全局布局组件的 `shared` 文件夹下使用 `index.ts` 统一导出所有的布局组件，示例代码如下：

```
// shared/index.ts
export * from './navbar/index';
export * from './toolbar/index';
```

- 将需要懒加载特性的内容存放在一个独立的文件夹中，如路由组件或其子组件，以及与之相关的其他资源和模块。
- 使用路由实现组件的懒加载，不要直接导入懒加载的目录。

综上所述，一个完整的应用结构示例如下：

```
src
├── app /** app 目录存放整个应用的文件 **/
│   ├── home /** 懒加载的组件独立存放在一个目录下 **/
│   │   ├── index.ts
│   │   ├── home.component.ts|html|css|spec.ts
│   │   └── ...
│   ├── login
│   ├── shared /** 共享模块 **/
│   │   ├── index.ts /** 使用 index.ts 文件统一导出 **/
│   │   ├── nav
│   │   │   ├── index.ts
│   │   │   └── ...
│   │   └── toolbar
│   │       ├── index.ts
│   │       └── ...
│   └── app.component.ts|html|css|spec.ts
├── assets /** 第三方库或其他资源存放在与 app 独立的文件夹中 **/
├── main.ts /** 应用入口文件 **/
└── index.html /** 应用入口页面 **/
```


18.2.4 Angular 模块约定

Angular 中涉及的模块主要包括根模块、特性模块、共享特性模块及核心特性模块四类。要保证每个应用至少有一个根模块，而且为了更好地定位和识别，应当考虑将根模块命名为 `app.module.ts`；同时每一个明显的特性都要有对应的特性模块，且将模块放到与特性同名的目录中；共享特性模块是指在 `shared` 目录中创建名为 `SharedModule` 的特性模块，且在该共享模块中声明那些可复用的组件、指令和管道等；区别于共享特性模块，还有一类只用一次的类，这些建议收集到核心模块中，让特性模块的结构更加简洁、清晰。创建名为 `CoreModule` 的核心模块，并存放到 `core` 目录下，并且坚持将共享给整个应用的单例服务放到 `CoreModule` 中。

18.2.5 组件相关约定

除了上述提到的部分关于组件的命名、格式、结构等，针对 Angular 的组件还有一些特定的约定和代码组织形式。

使用元素选择器的方式定义组件

在上述命名约定中提到，组件选择器应该使用“烤肉串”的形式命名。组件由指令派生而来，因此组件选择器支持元素、属性和其他的选择过滤方式。为了避免与指令定义混淆，建议定义组件都使用元素选择器的方式。因而组件选择器的命名采用与 W3C 草案中自定义元素的命名风格一致的“烤肉串”形式，示例代码如下：

```
@Component({
  selector: 'admin-user-list', // “烤肉串”命名形式
  templateUrl: 'admin-user-list.component.html'
})
export class AdminUserListComponent {}
```

将模板和样式文件分别抽离成单独文件

当模板或样式文件内容超过 3 行时就应该考虑抽离成独立文件。根据命名约定，模板文件名称应该为 `[component-name].component.html`，样式文件名称应该为 `[component-name].component.css`。

使用 @Input 和 @Output 装饰器

`@Component` 和 `@Directive` 装饰器元数据中的 `inputs` 和 `outputs` 属性应该使用 `@Input` 和 `@Output` 装饰器代替。`@Input` 和 `@Output` 装饰器应该放在与其装饰的属性声明的同一

行。同时要注意避免重命名输入输出属性，示例代码如下：

```
@Component({
  selector: 'user-button',
  template: '<button>{{label}}</button>'
})
export class UserButtonComponent {
  // 重命名容易导致混淆，应该避免
  // @Output('changeEvent') change = new EventEmitter<any>();
  // @Input('labelAttribute') label: string;

  // 推荐的方式
  @Output() change = new EventEmitter<any>();
  @Input() label: string;
}
```

这样做的好处是只在某处定义输入输出属性，当要修改属性名称时只需要修改属性定义的地方即可。如果在 `@Component` 和 `@Directive` 的元数据中定义了 `inputs` 和 `outputs` 属性，那么不仅需要修改属性名称，还需要修改 `inputs` 和 `outputs` 对应的值，而且在一个地方定义会让代码看上去更简洁、清晰、易读。避免重命名输入输出属性，也是为了避免名称的混淆和多次定义。

模板逻辑数据定义在组件类中

模板逻辑数据应该放在组件类中，而不是模板中。例如应该避免在模板中实现数学计算等复杂逻辑。

逻辑服务代码定义在服务中

组件应专注与视图相关的逻辑实现，其他逻辑应由服务来完成，以保持组件的简单，这也是单一职责原则的一种体现。从组件中独立出来的服务更易于代码复用和单元测试。

输出属性不要添加 `on` 前缀

事件名称不要添加 `on` 前缀，在事件的处理方法上添加 `on` 前缀。这样做一方面是为了与内置事件保持一致；另一方面是为了避免 `on-onEvent` 数据绑定表达式的出现。因为在 `Angular` 中从视图到数据源的单向数据绑定表达式不仅支持小括号的形式，还支持 `on-event` 的形式，如果给事件名称添加了 `on` 前缀，自然会导致 `on-onEvent` 形式的出现，

不免让人费解。示例代码如下：

```
<!-- 不好 -->
<user-edit (onUserSaved)="onUserSaved($event)"></user-edit>

// 不好
export class UserEditComponent {
  @Output() onUserSaved = new EventEmitter<boolean>();
}
```

而我们建议的应该是如下这样的写法：

```
<!-- 好 -->
<user-edit (userSaved)="onUserSaved($event)"></user-edit>

// 好
export class UserEditComponent {
  @Output() userSaved = new EventEmitter<boolean>();
}
```

实现接口来添加生命周期钩子

在 Angular 的运行生命周期中有很多内部事件产生，通常可以使用生命周期钩子在 Angular 运行过程中嵌入代码。添加生命周期钩子建议使用实现接口的方式，例如在组件初始化时添加钩子函数，示例代码如下：

```
export class HomeComponent implements OnInit {
  ngOnInit() {
    console.log('The home component initialized');
  }
}
```

如果不实现 `OnInit` 接口，编译器和 IDE 就不能给出适当的代码提示，假如 `ngOnInit()` 方法名称拼写错误会很难被发现。

18.2.6 指令相关约定

使用指令增强已有元素的功能

使用指令的目的在于增强已有元素的功能，当有表现层逻辑并且不需要模板时应该使用属性指令。使用属性定义指令的另一个好处是可以方便地在一个元素上应用多种属性指令。

使用 Host 装饰器替代 host 属性

使用 `@HostListener` 和 `@HostBinding` 装饰器代替 `@Component` 和 `@Directive` 装饰器的元数据中的 `host` 属性。这么做的原因跟在组件定义中使用 `@Input` 和 `@Output` 代替在 `@Component` 装饰器元数据中使用 `inputs` 和 `outputs` 属性是一致的。例如定义一个背景高亮指令，示例代码如下：

```
// 不好
@Directive({
  selector: '[highlight]',
  host: {
    '(mouseenter)': 'onMouseEnter()'
  }
})
export class HighlightDirective {
  onMouseEnter() {}
}

// 好
@Directive({
  selector: '[highlight]'
})
export class HighlightDirective {
  @HostListener('mouseenter') onMouseEnter() {}
}
```

18.2.7 服务相关约定

- (1) 同一个注入器中的服务应该是单例的，通常用来共享数据或者共用函数。
- (2) 服务应该遵循单一职责原则。
- (3) 应该在服务被共享的组件的最顶层组件注入器中注入服务。

在第 10 章依赖注入章节中，我们学习过 Angular 的依赖注入是分层级的，而服务本身又是单例，因此，当某个服务被一些组件共享时，该服务应该在这些组件的最顶层注入，才能达到共享数据或状态的目的。

- (4) 使用 `@Injectable` 类装饰器代替 `@Inject` 参数装饰器。

Angular 的依赖注入机制是根据类的构造函数参数类型声明来确定依赖关系的，因此当类的构造函数参数类型明确时，使用 `@Injectable` 类装饰器要比使用 `@Inject` 参数装

饰器精简得多。示例代码如下：

```
// 不推荐，使用参数注解代码冗长
export class UserComponent {
  constructor(
    @Inject(UserService) private userService: UserService,
    @Inject(Http) private http: Http) {}
}

// 推荐，使用类注解更简明
@Injectable()
export class UserComponent {
  constructor(
    private userService: UserService,
    private http: Http) {}
}
```

（5）独立数据服务。

在组件相关约定中提到了逻辑服务代码应该定义在服务中，其中所说的服务就包含数据服务，数据服务是用来处理诸如 XHR 调用、localStorage 操作、Cookie 操作等数据处理的逻辑；数据服务独立使得组件可以专注于与视图相关的处理逻辑，再者，独立的数据服务也为单元测试和模拟数据带来了方便，也更符合单一职责原则。

18.2.8 其他

除了单一职责原则、命名约定、编码约定、应用结构和 Angular 核心概念相关内容，还有一些非常有用的开发实践经验和小技巧，这里简要介绍如下。

使用 TypeDoc 帮助生成文档

TypeDoc 是一款 TypeScript 项目的文档生成工具，类似于 Java 语言的 JavaDoc。对于一个大型项目而言，文档固然重要，但通过手工方式维护文档是一件很费时费力的事情，通过 TypeDoc 将代码注释自动转换成文档则事半功倍。构建工具 Gulp 和 Grunt 都有 TypeDoc 的插件，自动构建文档非常方便。

使用 Snippets 代码片段工具

很多 IDE，例如 Visual Studio Code、Atom、Sublime Text 等，都有 Angular 的代码片段提示插件。通过输入关键字即可完成组件、服务、路由、管道等代码片段的创建。例如在 Visual Studio Code 中，输入 ng-component 在提示项中选择 Angular component snippet

并按回车键，编辑器就自动输入了一个完整组件类的代码，而且这些代码片段都是遵循最佳实践的，从而提升了开发效率。

关于 Angular 最佳实践的内容就介绍到这里。所谓无规矩不成方圆，严格遵循约定是写出具有良好可读性和可维护性代码的前提，同时，也是尽量减少代码错误的一种方式，希望这些最佳实践能给读者的 Angular 开发之旅带来帮助。

18.3 小结

通过前几章的实战，问卷调查系统的主体功能已经开发完成。首先，本章把重点放在项目构建上，它是工程化的重要步骤，通过代码质量检查、测试、打包、Docker 化等关键任务，能够保证项目的代码质量和风格，验证测试覆盖的功能，编译出易于分发的资源文件等，从而帮助开发者把高质量的代码发布到生产环境中；接着，本章给出了开发 Angular 应用的最佳实践，既包括了命名约定、编码约定、文件结构等方面，也有对 Angular 的重要部分如组件、指令和服务等开发方式的约定；最后，给出了一些开发中实用小技巧。

相信读者通过本章的学习，可以提高开发效率，编写出更高质量的代码。

第四部分

延伸篇

- 移动开发框架：ionic 介绍与实战
- 服务端渲染

19

移动开发框架：ionic 介绍与实战

19.1 移动开发

19.1.1 背景介绍

移动开发领域经历了十几年的快速发展，特别是在当前全面移动互联网的时代，进一步促进了该领域持续性的技术发展和创新，并且一直在印证着那个“前端摩尔定律”：前端每 18 个月会难一倍。

除了传统的 Mobile Web（HTML 5）和 Native，移动开发被注入了很多新的血液，如 Hybrid Web、React Native 等。随着可选择的增多，技术学习和选型的成本相应地也在提高，在系统性厘清各技术的特点、应用场景后，相信读者会豁然开朗，也能明白每种技术都有它存在的价值，并做到轻车熟路地结合所遇到的业务场景选择合适的开发模式，这也是大前端开发领域对工程师专业素养的要求之一。

作为本章的主角，ionic 经过多年的迭代发展，现已成为 Hybrid Web 阵营里成熟的解决方案之一。在重点介绍 ionic 之前，本章首先介绍移动开发领域的相关技术，让读者对移动开发的全局有一个整体认识。接下来介绍 ionic 并使用 ionic 快速开发本书的通讯录实例，最后介绍 ionic 的周边服务，使得读者对于该应用框架的生态环境有一个更全面的认识。

19.1.2 四种开发模式

在移动开发领域，目前主要有以下四种开发模式。

- **Mobile Web**: 一般指以 HTML 5、CSS 3 和 JavaScript 为基础，依托浏览器作为宿主运行环境的应用程序，日常简称 HTML 5 开发。
- **Hybrid Web**: 本质上也是 Mobile Web 页面开发，它以 WebView 渲染用户界面，以 JavaScript 作为基本逻辑，通过 JSBridge 等中间件方式跟底层客户端 API 进行双向通信，使得应用具备了客户端操作能力。典型的框架有 PhoneGap/Cordova、Sencha Touch 和 Framework 7 等。
- **React Native**: 近两年快速发展的开发框架。一方面，一般来说，它比 HTML 5、Hybrid 有更好的使用体验；另一方面，相比为各个平台（Android、iOS）各自开发一套 App，它的成本更低，且升级更新策略更灵活。除了 React Native，类似的框架还有 Weex、NativeScript 等。
- **Native**: 基于诸如 Java 或 Objective-C 编程语言直接使用手机系统 API 进行开发，这种开发方式就是日常所说的“原生开发”，典型的系统包括 iOS、Android、Windows Phone。

这四种技术的重要指标对比如表 19-1 所示。

表 19-1 移动开发模式对比

技术类型	迭代速度	用户体验	开发模式	更新机制	跨平台
Mobile Web	很快	还行	一次编写，到处运行	简单，直接发布	优
Hybrid	快	不错	一次编写，近乎到处运行	一般，可支持热更新	良
React Native	快	好	一次学习，多处编写	一般，可支持热更新	良
Native	慢	很好	各自开发，无法复用	复杂，部分平台支持热更新	差

19.1.3 技术选型

上一节介绍的不同移动开发模式的指标对比可以为技术选型提供参考依据。同时，下面列举的一些建议，相信能够辅助读者结合自己的业务场景选择相应的技术。

- 原型开发选择 Mobile Web 来快速试错，核心产品用 Native、React Native 来留存用户。HTML 5 开发成本低、发版容易、触达快，是产品试错的最佳选择。而 Native 优秀的渲染性能、用户体验保证了用户对产品的好感及二次使用。

- 创业团队适合采用 Mobile Web、Hybrid Web，大型团队适合综合采用多种技术。创业团队、小型团队的人力决定了多端开发的技术成本较高及产品质量难以保证。利用 Mobile Web 以及 Hybrid Web 的“一次编写，到处运行”优势更适合产品的快速发布。而对于大型团队，充足的人力及技术储备使得根据场景来选择合适的技术变成可能。“混合拳”能保证产品在用户体验与快速迭代之间维持恰当的平衡点。目前大多数大中型互联网公司都采用这种开发模式。
- 展示类应用选择 Mobile Web，对交互要求较高的应用则选择 Native。展示类场景对于性能要求不高，一般情况 Mobile Web 即可满足。而强交互类场景对于 UI 的渲染、重绘重排有较高要求，用 Native 能带来更佳体验。
- 访问硬件适合用 Native，信息展示则用 Web。一般的 Mobile Web 不具备摄像头、传感器、地理位置等调用硬件的能力。虽然可以利用 Hybrid Web 模式下提供的 Cordova 等插件来满足这些简单需求，但如果想做到性能更佳和适用范围更广，Native 才是最佳选择。
- 核心功能模块采用 Native 实现，周边辅助功能则用 Web，要把研发人力用在刀刃上。

需要注意的是，上述结论只是针对当下软硬件大环境下的一些共识。随着硬件性能的不断提升，页面渲染的速度会更快。同时，随着 PWA、WebAssembly、HTTP/2 等相关技术的继续发展，浏览器在缓存处理、消息通知、硬件处理、脚本执行、连接建立等方面会更上一个台阶，上述四种开发模式的权衡会不断地调整。不过正所谓“存在即合理”，这几种开发模式在短期内仍将继续共存并相互促进，不断寻找一个新的平衡点，供开发者选择。

19.2 ionic 平台介绍

19.2.1 概览

在 Hybrid Web 开发的解决方案中，ionic 可能是生态圈最完整、开发效率最高、社区反馈最好的解决方案。它是以 Angular 为基础，使用 TypeScript 语言开发的一套功能强大、跨平台、跨终端的 UI 框架。同时，借助于底层的 Cordova 平台，可以快速、方便地获得统一的 Native 操作能力，并通过简单的 CLI 命令即可轻松打包成 Android 或 iOS 等应用，做到一次开发，到处运行。其技术架构图如图 19-1 所示。

如图 19-1 所示，Angular、UI 组件、Cordova 插件组成框架的三要素。ionic 提供的 UI 组件基于 Web，并使用 Angular、TypeScript 等技术。这要求开发者具有基本的 HTML、

CSS、JavaScript 知识，同时需要掌握基础的 Angular 开发。建议读者通读本书第 5 章后再接着学习本章内容。

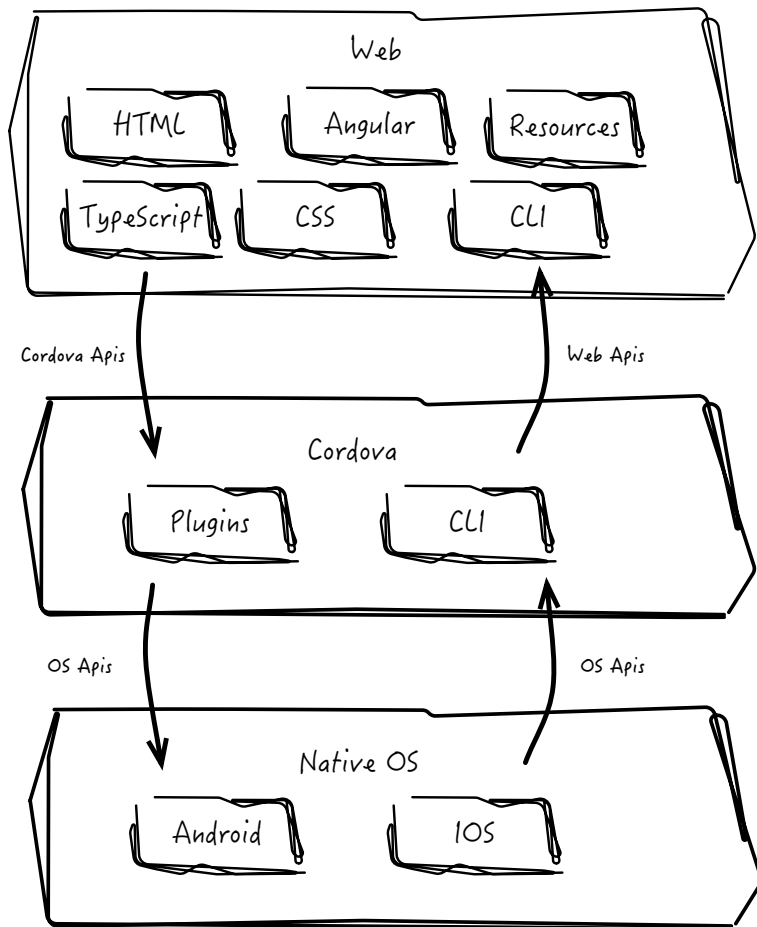


图 19-1 ionic 技术架构图

据 ionic 官方介绍，已经有超过 60 万的 App 是通过 ionic 来实现的，特别是新版的 ionic（当前最新版是 3.x）。得益于全新的 Angular 和 TypeScript 所带来的开发和性能的收益，相信有更多的人在 Hybrid Web 开发的选型上会着重考虑 ionic。本章介绍的 ionic 相关技术都是基于最新的 ionic 3.0 版本来讲解的。由于目前 ionic 也处于不断更新的阶段，一些知识点可能不是最新的，建议读者结合官方的更新记录来学习本章内容。

19.2.2 Cordova

Cordova 提供了一组统一、与平台无关的 API，这使得移动应用能够在 JavaScript 中以 API 形式访问原生的设备功能，如摄像头、麦克风等。开发者可以快速基于一套 HTML、CSS、JavaScript 代码进行多平台移动 App 开发。

Cordova 诞生已久。早在 2011 年 10 月，Adobe 收购了 Nitobi Software 和它的 PhoneGap 产品，然后宣布这个移动开发框架将会继续开源，并把它贡献给 Apache，由于商标原因更名为 Cordova。Cordova 是从 PhoneGap 中抽出的核心代码，是驱动 PhoneGap 的核心引擎，类似于 Webkit 与 Chrome 的关系。如图 19-1 所示，Cordova 架设在 ionic 底层，通过插件的形式给 Web 端提供 Native 的操作能力。ionic 维护了一个名为 ionic-native 的公共模块，封装了常用的 Cordova 插件，使得开发者更方便使用（后面有专门章节介绍）。接下来通过介绍 Cordova 与 JavaScript 端是如何通信的，来帮助读者更好地理解混合应用开发的运行机制。

通信原理

在 Android 系统中，WebView 组件的 `addJavascriptInterface()` 方法用于把 Java 对象注入到 JavaScript 的上下文中，使得 JavaScript 可以调用 Android 客户端的 Native 代码。示例代码如下：

```
class JsObject {
    @JavascriptInterface
    public String toString() { return "injectedObject"; }
}
webView.addJavascriptInterface(new JsObject(), "injectedObject");
webView.loadData("", "text/html", null);
webView.loadUrl("javascript:alert(injectedObject.toString());");
```

在 iOS 中，Cordova 往当前的 HTML 中插入一个不可见的 `iframe`，从而向 `UIWebView` 请求加载一个自定义的 URL，包括要调用的原生端的类名、方法名、参数、回调函数等信息。接着在加载过程中，`UIWebViewDelegate` 这个委托方法会被调用，定义如下：

```
- (BOOL)webView:(UIWebView*)theWebView shouldStartLoadWithRequest:(NSURLRequest*)
    request navigationType:(UIWebViewNavigationType)navigationType
```

通过解析参数，便可执行相应 JavaScript 端传过来的操作。另一种实现是 Cordova 通过发起特殊的 XHR 请求，被 Native 侧的 `NSURLProtocol` 拦截，同样通过解析参数实现方法的调用。



另一个角度是 Native 端如何调用 JavaScript 的方法，感兴趣的读者可参考 GitHub 上的 `WebViewJavaScriptBridge` 插件了解更多双向通信的知识。

下面将开始搭建开发环境，领略 ionic 良好的开发体验。

19.2.3 环境搭建

ionic 提供了两种创建方式。第一种是使用官网 ionic Creator 在线创建，网址为 <https://ionic.io/products/creator>，开发者注册账号后即可一步一步创建 App。其强大的编辑界面、拖曳式开发方式很适合产品经理、设计师等非开发人员快速构建产品的原型。感兴趣的读者可自行登录上述网址体验；第二种是使用 CLI 工具创建，这是程序员更为熟悉的模式。下面将以 CLI 这种方式来创建本章的 demo。

首先通过命令行工具全局安装 Cordova 及 ionic。

```
npm install -g cordova ionic
```



ionic 需要 Node 6.x、NPM 3.x 以上的版本，在本书的快速入门及问卷调查系统章节中都有专门的 Node 环境安装介绍，这里不再赘述，读者可以参考相关章节。

然后创建 ionic 项目。

```
ionic start myApp tabs
```

myApp 是应用名称；tabs 是应用模式，有 blank、tabs、sidemenu 三个可选值，分别代表空页面、底部导航、侧边栏三种界面风格。这里选择 tabs，这也是目前国内 App 的主流风格。



ionic 从发布开始就提供了 CLI 工具用于提供一致的项目模板，这也避免了开发者创建五花八门的项目结构和文件名，这种一致性给多项目开发团队带来的好处不言而喻。

接下来运行 App。

```
cd myApp  
ionic serve --lab
```

安装完成后即可使用 CLI 的 `serve` 命令直接运行应用，添加 `--lab` 参数来启动实验模式，可以方便模拟多个平台的运行效果。ionic 应用效果如图 19-2 所示。

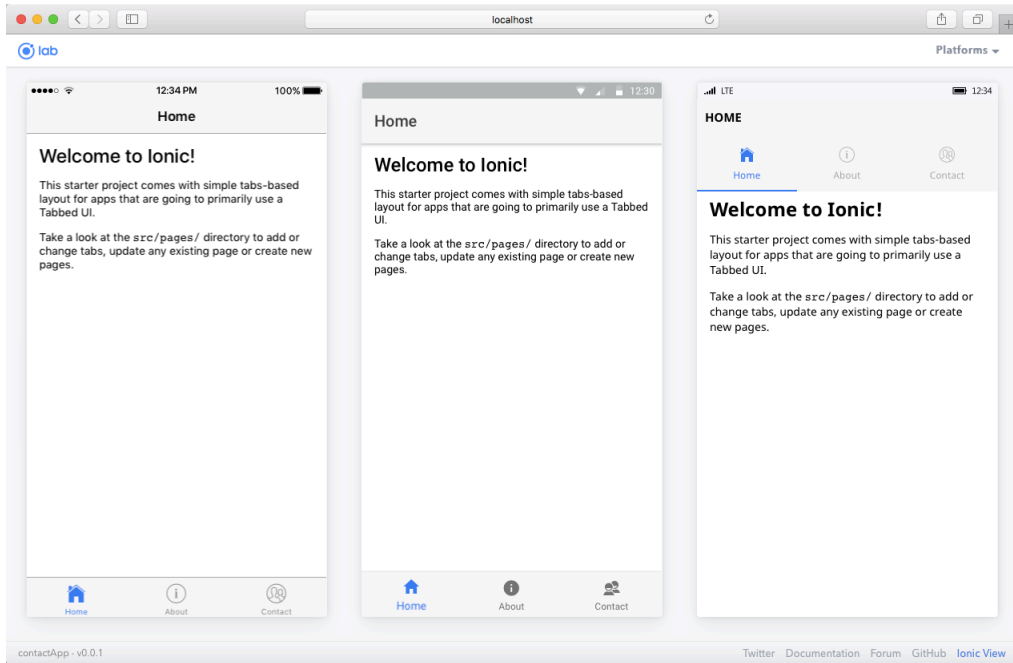


图 19-2 ionic 应用效果图

ionic 生成的脚手架包含了代码开发、Native 集成、构建打包、主题配置等部分，开发者只需要在对应的目录下继续添加文件即可。目录结构如下：

```

|—— config.xml           // App 配置文件，包括 App 名称、版本号、插件等
|—— hooks                // 说明文档
|   |—— README.md
|—— ionic.config.json    // 开发配置文件，可重写默认构建设置、添加代理等
|—— node_modules         // 项目依赖模块
|—— package.json         // 依赖配置文件
|—— platforms            // 生成客户端 App 的工程项目
|   |—— ios
|   |—— android
|—— plugins              // 项目中依赖的 Native 插件
|—— resources            // App 资源文件，包括图标、启动页图片等
|   |—— android
|   |—— icon.png

```

```
|   |—— ios
|   |—— splash.png
|—— src                      // 业务代码存放位置
|   |—— app                  // 启动页面模块
|   |—— assets               // 资源文件
|   |—— declarations.d.ts
|   |—— index.html           // 主页面
|   |—— manifest.json        // Service Worker 配置文件
|   |—— pages                // 项目页面
|   |—— service-worker.js    // 离线处理, 实现 PWA
|   |—— theme                // 主题文件
|—— tsconfig.json            // TypeScript 配置
|—— tslint.json              // Lint 设置
|—— www                      // 构建后文件存放位置
```



从 ionic 1 升级到 ionic 2 是一个巨大的工程, 从 ionic 2 开始不再使用 bower, 所使用的开发技术和底层预设语言也发生了颠覆性变化。一般情形下建议直接新建一个项目开发, 也可以参照官方的升级指引, 具体地址为: <http://ionicframework.com/files/Ionic2Migration.pdf>。

19.2.4 组件开发

我们知道, 用 Angular 开发的页面是由一个个组件组合而成的。简单来说, 页面开发即组件开发。配套丰富、完备的 UI 组件是 ionic 最大的亮点, 基于这些组件, 开发者可以快速构建各类场景的界面。它有以下几大特色:

- 数量众多。ionic 官方提供了接近 30 个组件, 加上强大的社区支撑, 开发者可以轻松找到满足自己需求的组件, 快速集成到已有的项目开发中。
- 种类丰富。从路由、页面等框架级别, 到列表展示、下拉刷新、上拉加载更多, 再到各种 UI 交互组件, 包括弹框、提示等, 最后是结合 App 唤起, 定位具体页面的 DeepLinker 等, ionic 给开发者提供了各种维度的组件清单。
- 跨平台展示。基于一套组件代码即可运行在各平台上, 真正做到一次编写, 到处运行。并且自动根据平台展示对应的样式风格, 比如在 Android 系统下默认带有 Material 风格的动画效果。
- 定制方便。得益于基于组件化的模式很容易集成, 同时结合 API 也更方便进行个性化定制。

按照功能划分，ionic 组件分类如图 19-3 所示。

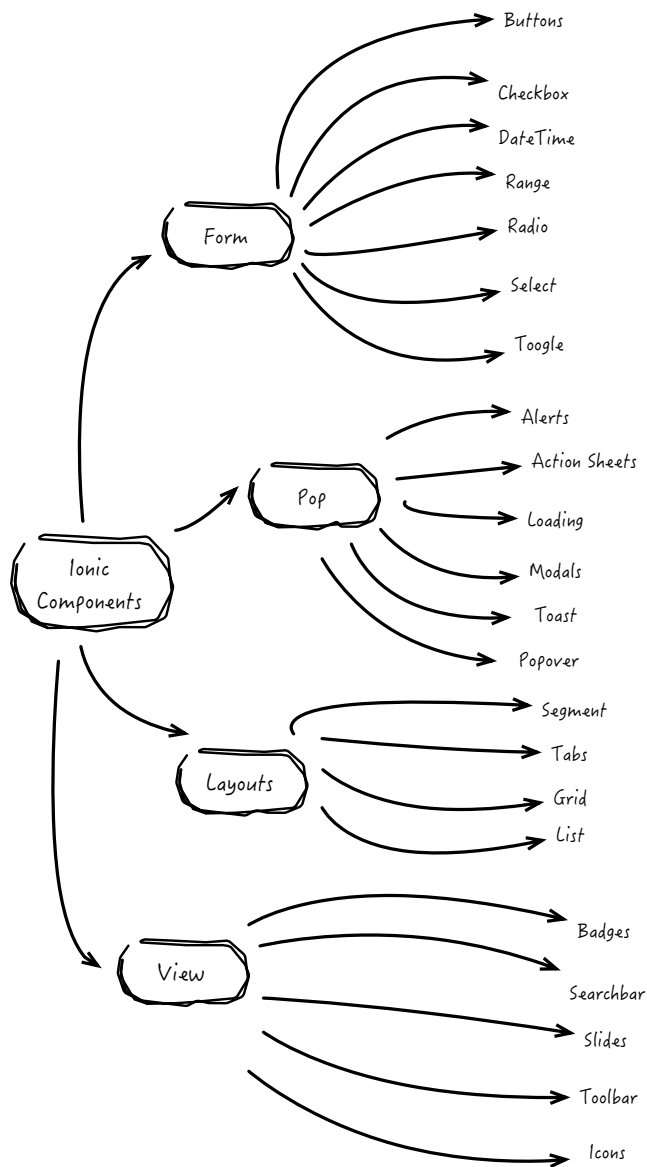


图 19-3 ionic 组件分类

本章重点在于系统地介绍 ionic 开发，并不会罗列各个组件的具体用法，后面有通讯录的实战案例，读者可以较直观地了解 ionic 的开发方式。

19.2.5 路由和导航

理解一个框架的路由、导航、跳转机制是进一步深入开发的重要基础。ionic 并没有简单复制 Angular 路由机制，而是参考了 Android、iOS 原生开发模式，定义了一套同样带有明确生命周期的路由系统，这便于开发者更简单、方便地利用框架暴露出来的事件进行数据初始化、缓存刷新、清理等操作。

ionic 的页面由导航控制器（Navigation Controller）集中管理，具体由 NavController 实现。跟导航相关的组件有 Nav、Tab 等，比如由 ion-nav 来管理的页面，像一个简单的栈，当页面被 Push 或被 Pop 时，即可实现向前导航或页面的历史栈后退，这与浏览器的 History API 类似，跳转过程如图 19-4 所示。

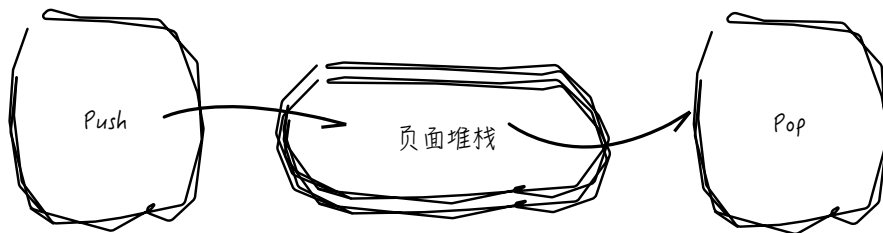


图 19-4 ionic 页面跳转过程

在上面创建的 myApp 实例中已经创建了默认的框架，使用 tabs 组件实现导航也很简单，可利用 root 属性指定初始化页面，这适合具有多个 tab 的 App 场景。具体代码如下：

```
// tabs.html
import { Component } from '@angular/core';
import { HomePage } from '../home/home';
import { AboutPage } from '../about/about';
import { ContactPage } from '../contact/contact';

@Component({
  templateUrl: 'tabs.html'
})
export class TabsPage {
  // this tells the tabs component which Pages
  // should be each tab's root Page
  tab1Root: any = HomePage;
  tab2Root: any = AboutPage;
```

```

    tab3Root: any = ContactPage;

    constructor() {

    }
}

<!-- tabs.html -->
<ion-tabs>
  <ion-tab [root]="tab1Root" tabTitle="Home" tabIcon="home"></ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="About" tabIcon="information-circle"></ion-
    tab>
  <ion-tab [root]="tab3Root" tabTitle="Contact" tabIcon="contacts"></ion-tab>
</ion-tabs>

```

上述代码中 `ion-tabs` 用来维护一组 `<ion-tab>`，每个 `<ion-tab>` 实际上都是一个导航控制器。这意味着每个 `tab` 都维护自己独立的页面历史栈，并且每个注入到 `tab` 里的 `NavController` 实例都和其他 `tab` 是不同的，如下面代码所示的 `home` 组件注入的 `navCtrl` 跟 `about` 组件里的 `navCtrl` 是不同的实例，在处理页面堆栈时要特别注意。

```

// home.ts
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';

@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})
export class HomePage {
  constructor(public navCtrl: NavController) {
  }
}

```

与 `<ion-tab>` 类似的还有 `<ion-nav>`，本质上它也是一个 `NavController`。在 `iOS` 系统中，由于没有物理返回键，每个页面都有一个返回箭头，来实现导航堆栈页面中的 `Pop` 操作。在 `ionic` 中已经为我们封装了一个 `<ion-navbar>` 组件，实现了返回键的功能，直接添加到模板文件中即可。代码如下：

```

<ion-header>
  <ion-navbar>
    <ion-title>Home</ion-title>

```

```
</ion-navbar>
</ion-header>

<ion-content padding>
  <h2>Welcome to ionic!</h2>
  <p>
    This starter project comes with simple tabs-based layout for apps
    that are going to primarily use a Tabbed UI.
  </p>
</ion-content>
```

有了 <ion-tabs> 和 <ion-navbar>, 整个应用的骨架就有了一个基本雏形。tabs 负责应用的骨架, navbar 负责应用的回退, 剩下的就是具体页面内的组件开发了。另外, 了解页面生命周期对于开发高性能的页面、理解框架的运行机制有很大的意义。ionic 中的组件同样是 Angular 组件, 除常规的组件生命周期事件外, ionic 还针对页面从离开当前页到进入新页面的过程添加了完整的钩子事件, 如表 19-2 所示。

表 19-2 ionic 页面钩子事件

事件名称	返回值	描述
ionViewCanEnter	boolean/Promise<void>	在页面激活前拦截处理, 适合做权限校验
ionViewWillEnter	void	当页面即将变为激活状态时触发
ionViewDidEnter	void	当页面已经变为激活状态时触发
ionViewDidLoad	void	只在页面加载完毕时触发一次, 当页面处于导航堆栈中, 被 Pop 后变成激活状态时也不触发, 适合做页面初始化
ionViewCanLeave	boolean/Promise<void>	在页面离开前拦截处理, 可在此时阻止
ionViewWillLeave	void	当页面即将变为非激活状态时触发
ionViewDidLeave	void	当页面已经变为非激活状态时触发
ionViewWillUnload	void	只在页面将被销毁时触发一次, 适合做清理工作

这些事件与 Android、iOS 等原生开发方式相似, 如图 19-5 所示。

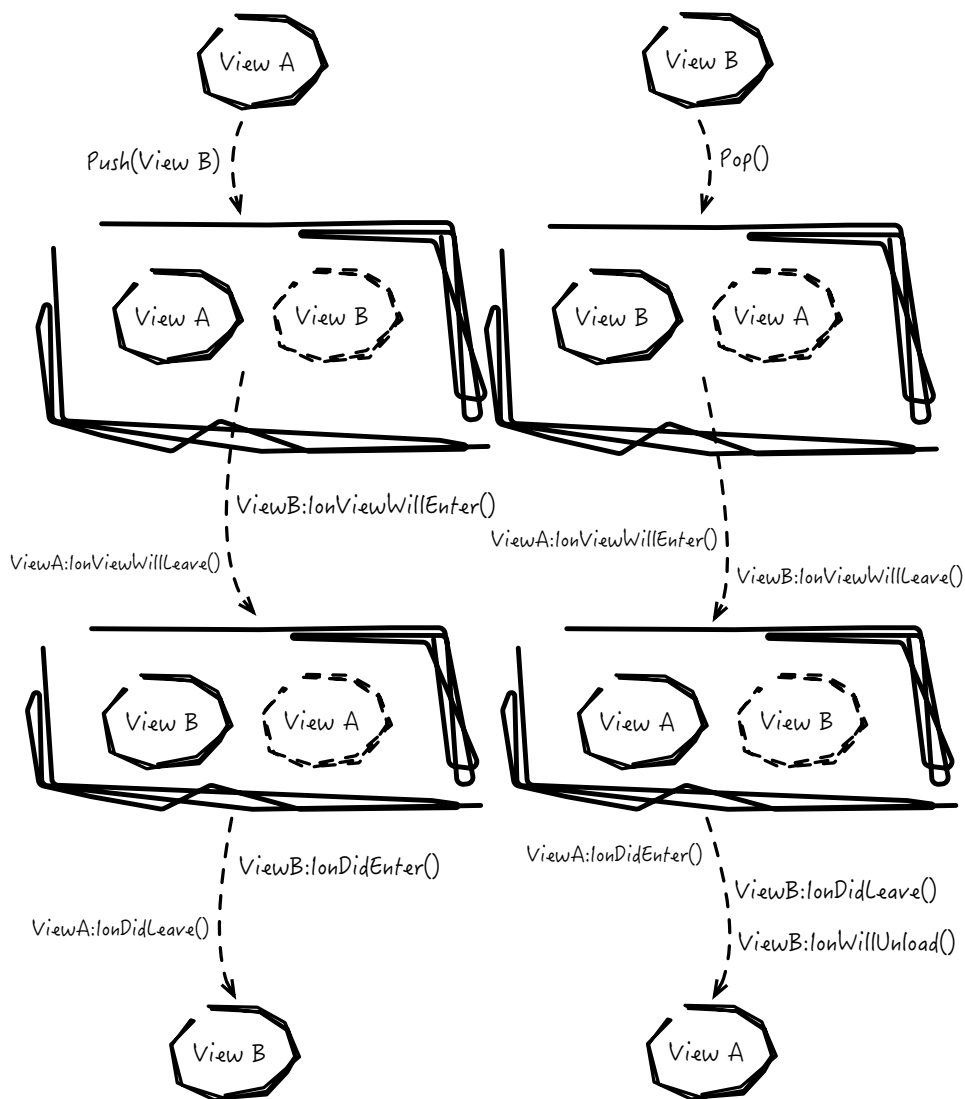


图 19-5 ionic 生命周期

上述生命周期方法可以让开发者细粒度地控制访问逻辑。比如在常见的权限验证场景中，只有具有相应权限的用户才能访问对应的页面，这一需求可以在`ionViewCanEnter`事件中校验，示例代码如下：

```
export class MyProfile {
  constructor(
    public navCtrl: NavController
```

```
    ){}

    ionViewCanEnter(): boolean{
        // 只有登录用户才能访问个人中心页面
        return isCurrentUserLogin()
    }
}
```

至此，ionic 组件开发中基础的页面部分已经介绍完毕，下面将开始介绍 ionic Native 技术，帮助我们的应用开启原生能力。

19.3 ionic Native

上文提到 Cordova 是一个开源的移动应用开发框架，借助于 Cordova 框架，ionic 具备调用原生代码的能力。Cordova 架起了 JavaScript 和原生代码之间相互调用的桥梁，在这个基础上构建了一套完善的体系，让我们可以以一种简单的流程开发 Hybrid 应用。

19.3.1 插件介绍

在 ionic 中通过独立的 ionic-native 包统一管理 Cordova 插件，它对常用的 Cordova 插件做了一层 TypeScript 包装，添加了类型定义，使得开发者可方便地将其集成到应用开发中。同时，为了提高开发体验，ionic-native 对回调做了 Observable/Promise 封装。表 19-3 列举了一些常用的插件。

表 19-3 ionic-native 常用插件

插件名称	描述	插件名称	描述
Alipay	支付宝支付	App Version	App 版本信息
App Update	App 版本更新	Camera	摄像头
Clipboard	粘贴板	Code Push	资源热更新
Contacts	手机通讯录	Deeplinks	自定义资源协议
Dialogs	客户端弹框	File	文件传输
FileOpener	文件打开	Geolocation	定位
Image Picker	图片选择	In App Browser	在 App 中打开外链
Keyboard	软键盘	Network	网络条件
Photo Library	相册	Status Bar	状态栏



欲了解更多的插件，读者可以在 <http://ionicframework.com/docs/native/> 官网或 GitHub 上搜索

19.3.2 插件使用

首先安装 ionic-native，命令如下：

```
npm install @ionic-native/core --save
```

然后安装具体的插件。安装 ionic-native 插件的命令类似于 npm 安装命令，如下所示：

```
ionic cordova plugin add cordova-plugin-file  
npm install @ionic-native/file --save
```

在代码中，可直接导入并调用对应的方法。代码如下：

```
import { File } from '@ionic-native/file';  
  
constructor(private file: File) { }  
// ...  
this.file.checkDir(this.file.dataDirectory, "mydir")  
  .then(_ => console.log("Directory exists"))  
  .catch(err => console.log("Directory doesn't exist"));
```

值得注意的是，除了 ionic-native 中提供的插件，还有很多第三方插件可供使用，在项目中使用时，可能会遇到一些问题。由于 TypeScript 是强类型语言，ionic-native 中的插件已经包含了类型文件，描述该插件 API 的详细信息，而很多插件并未有相应的处理，直接暴露在 window 或 window.plugins 上，比如下面的调用方式会报错。

```
window.plugins.somePlugin.doSomething();  
// 或者  
someGlobal.doSomething();
```

// 编译错误：error TS2339: Property 'plugins' does not exist on type 'Window'.

一种解决方法就是把 window 定义为 any 类型，绕过编译器的检查。代码如下：

```
(<any>window).plugins.somePlugin.doSomething();  
// 或者  
declare var window:any;
```

在 ionic 中，更好的解决方法是往 `declaration.d.ts` 中添加插件类型说明，因为将 `window` 设置为 `any` 导致失去了类型校验功能，我们只需要把 `plugins` 作为 `window` 的一个属性即可。如果有其他的全局对象，则建议也在此文件中集中维护。

```
// declaration.d.ts

interface Window {
  plugins: () => void; // 给 window 添加 plugins 属性
}

declare var someGlobal: any; // 定义全局变量
```

19.3.3 插件开发

当已有的 Cordova 插件不能满足需求时，就需要自行开发插件了。Cordova 插件有对应的模板，参照模板添加对应的代码与配置即可。下面是 Cordova-plugin-wechat 插件的目录结构：

```
├── README.md
├── plugin.xml // 插件配置文件
├── src // 不同平台的 Native 代码
│   ├── android
│   └── ios
├── www
│   └── wechat.js // 暴露给 JavaScript 端的接口
```

编写插件需要具备原生开发能力，需要为不同平台开发相应的插件并封装给前端使用。

19.4 样式和主题

19.4.1 平台样式

ionic 配备的平台适配风格使得开发者可以构建丰富、美观的产品，同时它也提供了简易的方式方便个性化定制。对于框架提供的每一个组件，ionic 已默认添加 iOS、Android、Windows Phone 平台相应的样式。这节省了很多 UI 处理时间，不过同时也存在一个隐含的问题，毕竟很多产品的原型设计都只有一套，ionic 提供的默认风格未必符合要求，这也给开发者带来额外的覆写样式成本。下面将介绍这些细节，让开发者可以打造一款属于自己风格的产品。

对于每一个平台,ionic 默认提供了一套跟该操作系统风格一致的样式,如在 Android 中是 Material 模式 (这是 Android 5.0 开始使用的系统风格), 平台模式通过 CSS 来控制。示例代码如下:

```
<ion-app class="ios">
```

各平台的默认样式如表 19-4 所示。

表 19-4 各平台的默认样式

平台	模式	描述
iOS	ios	任何 iOS 设备, 如 iPhone、iPad、iPod、iWatch 等
Android	md	任何安卓设备, 如平板电脑、手机等
Windows Phone	wp	Windows 设备
其他	md	非上述平台默认用 md 模式

ionic 在 App 启动时根据平台信息注入更多的样式, 使得开发者可以更深入地对具体版本进行控制处理。在 iOS 中默认会注入的样式如下:

```
<ion-app class="ios platform-mobile platform-ios platform-ios9 platform-ios9_1 platform-iphone platform-mobileweb">
```

开发人员可以在 AppModule 中全局修改平台模式, 比如设置在所有平台下都使用 iOS 模式, 示例代码如下:

```
import { IonicApp, IonicModule } from 'ionic-angular';

@NgModule({
  declarations: [ MyApp ],
  imports: [
    IonicModule.forRoot(MyApp, {
      mode: 'ios' // 所有平台都使用 iOS 风格
    }, {}
  ],
  bootstrap: [IonicApp],
  entryComponents: [ MyApp ],
  providers: []
})
```

上面的代码在应用模块初始化时通过传入 mode 属性改写了平台样式。



细心的读者应该能注意到 `mode` 被包裹在一个对象里面，这意味着还有其他全局属性可以在这里重设。实际上 `ionic` 在这里提供了一个 `Config` 类，开发者可以重写全局默认配置，包括 `backButtonText`（回退按钮文本提示）、`iconMode`（图标模式）等。对于 `mode` 属性，在 `ion-app` 初始化时通过 `this.setElementClass(this._config.get('mode'), true);` 来重写样式。同时，还允许开发者通过 `get/set` 方式自定义全局配置，在应用中的任何地方都可以读取到这个配置项。

19.4.2 主题

主题风格代表一款产品的定位，一般在产品设计之初就会考虑并稳定下来。然而在某些场景中，特别是在电商类 App 中，各类促销活动通常会切换整个平台的主题。`ionic` App 支持主题风格变更，只需要调整代码中 `src/theme/variables.scss` 文件的 `$colors` 变量映射即可达到切换效果。示例代码如下：

```
$colors: (  
  primary:    #008aff,  
  secondary:  #32db64,  
  danger:     #f53d3d,  
  light:      #f4f4f4,  
  dark:       #333,  
  favorite:   #698b7b,  
  toolbar:    #2b5d82  
);
```

如上所示，修改相关变量即可快速切换主题，其中 `primary` 是必须重写的，这样 `ionic` 组件将使用这种颜色作为默认值。同时，`ionic` 允许开发者添加自定义颜色到上述 `$colors` 映射中，并可以在组件中更方便地使用。比如：

```
$colors: (  
  // ...  
  
  myColor: #55acee, // 自定义颜色  
  
  // 或者自定义基本色跟对比色  
  myColor:(  
    base: #55acee, // 元素的背景色
```

```
        contrast: #ffffff // 文本颜色
    )
};
```

这样，在组件中即可使用这种自定义颜色了。示例代码如下：

```
// 通过属性设置
<button color="myColor">I'm a button</button>

// 或者在 SCSS 代码中设置
my-component {
  background : color($colors, myColor)
  background : color($colors, myColor, base)
}
```

ionic 中的很多组件都有 color 属性，开发者可以自定义颜色。而在 SCSS 中，ionic 提供了内置函数 color() 允许开发者获取自定义颜色值。color 函数的源码定义如下：

```
// 获取 color 映射值
// @param {Map} $map - 映射变量集合，如 $color
// @param {String} $color-name - 变量名，如 myColor
// @param {String} $color-key - 基本色 / 对比色（可选，默认为基本色）
// @return {Color} - 定义的颜色值，#55acee
function color($map, $color-name, $color-key:null) {
  // 具体实现
}
```

这样，开发者可以灵活地设计多套主题风格，并根据具体场景进行相应的切换，从而轻松提升产品的运营能力。

19.4.3 全局变量

ionic 的样式是基于 SASS 开发的，在更细致的组件级别设计中，ionic 定义了上百个全局变量，用于定义组件的样式细节。这有利于样式的统一维护。同时，ionic 也提供了重写机制，使得开发者可以自定义开发。下面列举几个 text 相关属性。

表 19-5 text 属性

变量名	默认值
\$text-color	#fff
\$text-input-highlight-color-valid	#32db64
\$text-input-highlight-color-invalid	#f53d3d

对于开发者来说，在 `src/theme/variables.scss` 中重写上述变量值后，所有组件对应的该属性样式值也会相应变化。另外，开发者也可以在该文件中定义自己的变量值，达到一处维护，到处更新的效果。示例代码如下：

```
// src/theme/variables.scss

$text-color: #333; // 重写平台变量
$my-control-height: 40px; // 自定义变量


// my-component.scss
.header {
    height: $my-control-height;
}
.sub-header {
    height: $my-control-height;
}
```

这并不是 ionic 的特有功能，对于基于 SASS 开发的应用来说，这都是常用的最佳实践。

19.4.4 工具属性

除此之外，为了提高样式的开发效率，ionic 提供了常用的工具属性，开发者可以直接作为属性添加到组件中，来完成一些常用的样式修改。下面列举一些常用的工具属性。

文本变化

文本变化属性如表 19-6 所示。

表 19-6 文本变化属性

属性名称	描述
text-left	文本左对齐，其他对齐方式还有：text-center、text-right、text-justify
text-wrap	文本包裹，其他方式还有：text-nowrap
text-lowercase	文本小写，同样还有大写、首字母大写：text-uppercase、text-capitalize

元素控制

元素控制属性如表 19-7 所示。

表 19-7 元素控制属性

属性名称	属性值
padding 属性	padding、padding-top、padding-left、padding-right、padding-bottom、padding-vertical、padding-horizontal、no-padding
margin 属性	margin、margin-top、margin-left、margin-right、margin-bottom、margin-vertical、margin-horizontal、no-margin
ion-buttons 对齐方式	start、end、left、right
ion-checkbox 对齐方式	item-left、item-right

开发者可以在组件开发中直接使用上述属性，以快速实现样式控制。示例代码如下：

```
<my-component no-padding text-left></my-component>
```

19.4.5 Iconfont

Iconfont 用字体文件取代图片文件来展示图标、特殊字体等元素。由于使用的是矢量图，在响应式页面中不会随着尺寸的变化而失真，Iconfont 在实际的开发中被经常使用。当然也存在兼容性、文件过大等问题，读者可根据自己的场景选择 Iconfont、雪碧图、base64、Canvas 等各种图片方案。

ionic 提供了一套矢量图标库，默认在 src/theme/variables.scss 中，通过 @import "ionic.ionicons" 导入。开发者可直接在 HTML 代码中使用，如 <ion-icon name="add"></ion-icon>，通过指定 icon 的 name 属性即可使用。

大多数图标具有 iOS、iOS-Outline、Material Design 三种样式，且会根据平台自动适配。如果需要对不同的平台指定样式，则需要使用 md 和 ios 属性，如<ion-icon ios="ios-add" md="md-add"></ion-icon>。



欲了解更多的图标文件,可到<http://ionicframework.com/docs/v2/ionicons/>查看。

通过 ionic 提供的主题、平台样式、变量、工具属性、Iconfont 等解决方案，开发者可以游刃有余地打造出具有自己特色的产品。

19.5 ionic CLI

在 19.2.3 节中提到过 ionic CLI，但只列出简单的使用示例，下面会对其进行详细介绍。

ionic CLI (Command Line Interface) 是开发 ionic App 过程中使用的主要工具，它包含了一些常见的操作命令，涵盖了从项目创建、开发、调试、打包、签名到发布等生命周期所需要的工具及命令的集合。熟练掌握 CLI，对于 ionic 开发者来说是基础而重要的事情。

安装 CLI 需要 Node 6.0+ 和 NPM 3+ 的环境，所需环境安装好即可运行下面命令安装 ionic CLI。

```
npm install -g ionic
```

安装完成后，可以运行 `ionic info` 命令来查看本地的 ionic CLI 信息。笔者的本地开发环境的 CLI 信息如下：

```
cli packages: (/xxx/lib/node_modules)
  @ionic/cli-utils : 1.15.2
  ionic (Ionic CLI) : 3.15.2
local packages:
  @ionic/app-scripts : 3.0.1
  Ionic Framework    : ionic-angular 3.8.0
System:
  Node : v6.9.4
  npm  : 3.10.10
  OS   : macOS Sierra
Misc:
  backend : pro
```

随着 ionic 的不断迭代，在升级过程中若遇到一些兼容性问题，则可以通过 `info` 命令来排查是否是版本问题。

下面介绍 CLI 的常用命令。

创建

start 命令用于创建一个新的 ionic 项目，在上面的章节中已使用过。它默认使用 Tabs 项目模板，除此之外，还有 blank、sidemenu 等其他选项。

```
ionic start myApp blank
```

开发

generate 命令可以帮助开发者快速生成 ionic 推荐的模板文件，包括 Component、Directive、Page、Pipe、Provider、Tabs。创建一个新页面的命令如下：

```
ionic generate page myList
```

此时将会在 src/pages/my-list/ 目录下生成几个文件，如 my-list.html、my-list.ts、my-list.scss、my-list.module.ts 等。

调试

serve 是最常用的命令，可以在开发和测试过程中启动一台本地开发服务器，结合 livereload 监测文件变化及 Chrome 的 devtool 等工具，可以实时开发、调试、下断点，非常方便。命令如下：

```
ionic serve
```

同时，还可以添加其他参数，更细致地配置调试细节。如表 19-8 所示是常用的几个调试参数设置。

表 19-8 常用的调试参数设置

参数	描述
[--consolelogs -c]	打印 Console 日志
[--port -p]	指定端口，默认为 8100
[--all -a]	监听所有 IP 地址（0.0.0.0）
[--lab -l]	在多窗口中模拟多平台测试

run 命令将 App 部署到特定的设备上，在后期真机调试或 ionic Native 调试时很实用。命令如下：

```
ionic cordova run ios --device
ionic cordova run ios --list
```

```
ionic cordova run ios --release --buildConfig=myBuildConfig.json --target=iPhone-5
```



跟 `serve` 命令不同, `run` 命令是 Cordova 提供的, 需要以 `ionic cordova xxx` 形式使用。

添加 `--livereload` 选项开启 live reload 功能, 这样编译后的 Hybrid 应用就会监测任何文件的改变并在需要的时候重新载入 App, 减少了频繁重新编译 App 的时间。注意, 为了 live reload 能够正常工作, 开发机和模拟器必须在一个相同的本地网络中, 并且设备必须支持 WebSocket。跟 `serve` 命令类似, 同样可以增加 `port`、`address`、`serverlogs` 等属性赋值调试。

`emulate` 命令用于将 App 部署到模拟器上。命令如下:

```
ionic cordova emulate
ionic cordova emulate ios
ionic cordova emulate android
```

`plugin` 命令用于管理 Cordova 插件, 可以查看、安装、移除相关插件。项目根目录下的 `config.xml` 文件里面的 `plugin` 节点, 用于显示当前 ionic 项目用到的相关插件。与 `plugin` 相关的参数有 `add`、`list`、`search`、`remove`, 具体代码如下:

```
ionic cordova plugin ls
ionic cordova plugin add cordova-plugin-camera@^2.0.0
ionic cordova plugin add https://github.com/myfork/cordova-plugin-camera.git#2.1.0
ionic cordova plugin add ../cordova-plugin-camera
ionic cordova plugin add ../cordova-plugin-camera.tgz
ionic cordova plugin rm camera
```

打包

`resources` 命令用于自动生成 App 的图标及启动页面。在根目录下建立一个 `resources` 文件夹, 把设计好的 `icon`、`splash` 文件放进去, 执行该命令后 ionic 将生成各尺寸的图标与启动页面。命令如下:

```
ionic cordova resources
ionic cordova resources android
ionic cordova resources -i
```

注意, 支持的图片格式有 PNG、AI 和 PSD, 同时, 图标文件的尺寸必须为 192 像素 × 192 像素, 启动页面的尺寸则必须为 2208 像素 × 2208 像素。

`platform` 命令用于对平台代码进行管理，生成相应的项目脚手架，其中 `ls` 参数用于查看当前平台支持情况。具体代码如下：

```
ionic cordova platform ls

# Available platforms:
# android ~6.3.0
# blackberry10 ~3.8.0 (deprecated)
# browser ~5.0.0
# ios ~4.5.1
# osx ~4.0.1
# ubuntu ~4.3.4 (deprecated)
# webos ~3.7.0
# windows ~5.0.0
# www ^3.12.0
```

可以用 `add` 参数来添加特定的平台。比如：

```
ionic cordova platform add android
ionic cordova platform add ios
```

同样，可以用 `rm` 进行移除，用 `update` 进行更新。

`build` 命令由 `prepare` 与 `compile` 两个子命令组成，用于生成指定平台的应用。在 `prepare` 阶段，执行的操作包括把 `config.xml` 转换为指定平台的 `manifest` 文件，拷贝图标，启动图片、插件，为编译阶段做准备。在 `compile` 阶段，生成指定平台的 App，可以指定模式或添加签名文件，生成 `release` 包。比如：

```
ionic cordova build android --debug --device
ionic cordova build android --release --buildConfig=myBuildConfig.json
ionic cordova build android --release -- --keystore="android.keystore" --
storePassword=android --alias=mykey
```

还有其他一些命令，读者可以通过 `ionic help` 进一步了解。

19.6 通讯录实例

本节将结合上述章节中介绍的知识，使用 `ionic` 开发简单的通讯录例子，让读者对于 `ionic` 开发有更深入的理解。

通讯录的功能包括查看联系人列表、查看通话记录、查看收藏联系人列表、查看联系人明细、收藏、拨打电话等，最终运行效果如图 19-6 所示。

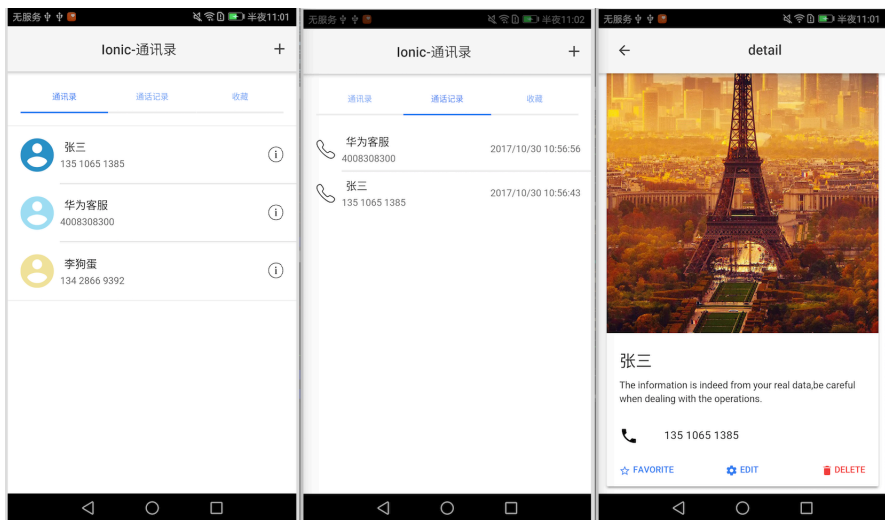


图 19-6 ionic 通讯录例子

19.6.1 项目搭建

通过前文介绍，我们知道使用 ionic CLI 可以极大地提高开发效率。在命令行通过 `ionic start contactsDemo` 即可方便地创建一个项目，这里并没有加上 `tabs` 参数，是因为通讯录只需要一个 tab 即可满足显示需求。项目的目录如下：

```

├── config.xml
├── hooks
├── ionic.config.json
├── node_modules
├── package.json
├── platforms
├── plugins
├── resources
├── src
│   ├── app
│   │   ├── app.component.ts
│   │   ├── app.html
│   │   ├── app.module.ts
│   │   ├── app.scss
│   │   └── main.ts
│   ├── assets
│   └── icon

```

```

| | | | |   └── favicon.ico
| | | | |─── declarations.d.ts
| | | | |─── index.html
| | | | |─── manifest.json
| | | | |─── pages
| | | | |   └── home
| | | | |       ├── home.html
| | | | |       ├── home.scss
| | | | |       └── home.ts
| | | | |─── service-worker.js
| | | | |─── theme
| | | | |   └── variables.scss
| | | | |─── tslint.json
| | | | |─── www

```

对主目录的说明在上面创建 ionic 项目的章节中已经介绍过，这里不再赘述。下面将开始代码的编写。

19.6.2 主页面

home 作为主页面显示通讯录的内容，考虑到有通讯录、通话记录、收藏三个列表项，ionic 的 Segment 组件很适合这种场景。模板代码如下：

```

// home.html
<ion-header>
  <ion-toolbar text-center>
    <ion-title> ionic-通讯录 </ion-title>
    <ion-buttons end>
      <button ion-button icon-only (click)="toManage()">
        <ion-icon name="add"></ion-icon>
      </button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content>
  <div >
    <ion-segment [(ngModel)]="type">
      <ion-segment-button value="contact">
        通讯录

```

```

</ion-segment-button>
<ion-segment-button value="dial">
    通话记录
</ion-segment-button>
<ion-segment-button value="favorite">
    收藏
</ion-segment-button>
</ion-segment>
</div>

<div [ngSwitch]="type">
    <ion-list *ngSwitchCase="'contact'">
        <ion-item *ngFor="let item of deviceContacts0bs | async ; let i=index" (click)=
            "call(item)">
            <ion-avatar item-left>
                <ion-icon name='ios-contact-outline' item-left [ngStyle]='{'color':
                    getRandomColor(i)}' style="font-size:60px"></ion-icon>
            </ion-avatar>
            <h2 style="margin:5px;">{{item.displayName}}</h2>
            <p>{{ showPhone(item) }}</p>
            <ion-icon name='ios-information-circle-outline' item-right (click)="toDetail(
                item,$event)"></ion-icon>
        </ion-item>
    </ion-list>

    <ion-list *ngSwitchCase="'dial'">
        <!-- 通话记录的模板内容 -->
    </ion-list>

    <ion-list *ngSwitchCase="'favorite'">
        <!-- 收藏列表的模板内容 -->
    </ion-list>
</div>
</ion-content>

```

模板代码基本上囊括了业务代码的主要部分，其包括下面这些组件。

- **ion-header**: ion-header 组件是页面的默认组成部分，显示统一的头部信息，可以嵌套 nav-bar、ion-toolbar 等组件进行更详细的自定义设计。
- **ion-toolbar**: ion-toolbar 是通用的工具栏组件，一般用于构建头部、底部等页面元

素。它默认用 flex 布局，可以在里面添加具体的控件元素，如 icon、button、label 等，这里通过 text-center、end 来辅助快速实现子元素的定位。

- **ion-icon**: ionic 提供了丰富的字体图标，通过 ion-icon 组件可以方便地添加。比如在头部添加 add 图标，点击可以创建新联系人。
- **ion-content**: ion-content 组件是另一个页面的默认组成部分，显示页面的主体部分。
- **ion-segment**: ion-segment 组件用于分段展示内容，类似于页面内的 tabs 控件，里面嵌套 ion-segment-button 组件来显示标题，可以通过 segmentValue 变量来控制显示或隐藏哪部分内容。这里通过绑定 type 这个属性，可选的值为 contact、dial、favorite，点击 ion-segment-button 实现 segment 样式的选中切换。通过 ngSwitchCase 指令，实现三个列表内容的切换。
- **ion-list**: ion-list 组件是衡量组件质量的试金石。ionic 的 List 模板众多，使用方便。本例中通讯录列表用了 Avatar List、通话记录列表用了 Icon List、收藏列表用了 Thumbnail List。除此之外，配合 ion-refresher 组件可以实现下拉刷新，配合 ion-infinite-scroll 组件可以实现上拉加载更多，配合 virtualScroll 指令可以实现列表的 DOM 复用，提高页面的渲染性能。这些组件满足了列表的各种场景，提高了开发者的开发效率，也降低了代码的维护成本。

在页面逻辑部分，主要显示三个 segment 下对应的列表数据，具体如下。

- **通讯录列表**: 通讯录数据来源于手机，可以通过 cordova-plugin-contacts 这个开源插件来获取手机上的通讯录数据。在控制台上输入下面命令进行插件的安装。

```
ionic cordova plugin add cordova-plugin-contacts
npm install --save @ionic-native/contacts
```

安装完插件后，即可通过暴露在全局变量 navigator 中的 contacts 属性获得通讯录内容。注意，需要在 deviceready 事件触发后才能获得到。代码如下：

```
this.deviceContactsObs = new Observable(observer=>{
  this.platform.ready().then(()=>{
    navigator['contacts'] && navigator['contacts']
      .find([navigator['contacts'].fieldType.displayName],
        (list)=> {
          observer.next(list);
          this.chRef.detectChanges();
        },
```

```

        this.errorHandler);
    });
});

```

这里返回一个 Observable 对象，结合模板中的 async 异步管道，实现模板的自动订阅并更新界面。

- **通话记录列表：**通话记录在每次拨打电话时记录下来并展示。ionic 提供了 storage 对象，适合做简单的客户端持久化存储。考虑到收藏列表也需要用到 storage 对象，所以这里封装了一个 contact.service，用于通讯录的存储操作。代码如下：

```
// contact.service
```

```

import { Injectable, ChangeDetectorRef } from '@angular/core';
import { Platform } from 'ionic-angular';
import { Storage } from '@ionic/storage';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class contactService {
    public favoriteContacts = []; // 收藏列表
    public diallList = [];       // 通话记录列表

    constructor(
        private platform: Platform,
        public storage: Storage
    ){
    }

    async initData() {
        await this.storage.ready();
        this.favoriteContacts = await this.storage.get('favoriteList') || [];
        this.diallList = await this.storage.get('diallList') || [];
    }
}

```

在 initData() 方法中从 storage 里获取收藏列表及电话拨打记录，并直接绑定到展示列表上。

- **收藏列表：**使用 favoriteContacts 数组显示收藏记录，数据获取方式同上。

除了数据绑定，主页面的其他逻辑比较简单，这里直接列出代码。

```
// home.ts
constructor(public navCtrl: NavController
              ,public domSanitizer: DomSanitizer
              ,public chRef: ChangeDetectorRef
              ,private storage: Storage
              ,public contactService: contactService
              ,private platform: Platform) {

// 显示手机号
showPhone(item) {
    return item.phoneNumbers? item.phoneNumbers[0].value:'no record';
}

// 显示头像
showAvator(item) {
    return item.photos?
        this.domSanitizer.bypassSecurityTrustUrl(item.photos[0].value):'./assets/
        contact.jpg';
}

// 拨打电话
call(item){
    if(item.phoneNumbers && item.phoneNumbers[0].value) {
        item.timestamp = + new Date();
        this.contactService.dialList.splice(0,0,item);
        this.contactService.storage.set('dialList',this.contactService.dialList
        );
        window.open(`tel:${item.phoneNumbers[0].value}`,'_system');
    }
}

// 跳到详情页面
toDetail(item,events) {
    events.stopPropagation();
    this.navCtrl.push(DetailPage,{item:item});
}
```

```
// 跳到添加（编辑）页面
toManage() {
    this.navCtrl.push(ManagePage);
}

// 依次显示小清新颜色数组
getRandomColor(i) {
    return ['#0092c7', '#9fe0f6', '#f3e59a',
            '#f3b59b', '#f29c9c', '#f8e400',
            '#f26378', '#f13dbad', '#ff7d48',
            '#a2ef57'][i%9];
}
}
```

- **showAvatar()**: DomSanitizer 是 Angular 为了防止页面受到 XSS 攻击所提供的“净化”服务，如加载外部资源时默认被阻止，则需要通过 `bypassSecurityTrustUrl()` 方法来允许加载。
- **call()**: 该方法包含把通话记录添加到 `storage` 中和拨打电话两个逻辑。前者通过 `storage.set()` 持久化到客户端中，后者通过 `tel://手机号` 伪协议拨打电话，ionic 默认已添加对该协议的支持。
- **toManage()**: 跳转到详情页面，通过 ionic 自带的导航机制 `NavController.push` 跟 `NavController.pop` 实现控制。
- **getRandomColor()**: 用一组小清新颜色随机显示通讯录头像的背景色，这比随机生成颜色的视觉效果更佳。

至此，主页面的逻辑就介绍完了。其他功能模块相对简单，读者可以到 <https://github.com/angular-programming/ionic-contacts-demo> 上查看完整的项目代码。

19.7 小结

本章系统地介绍了 ionic 的开发与实战。首先分析了当前移动开发领域的现状；接着全局性介绍了 ionic 框架的架构、组件分析、导航原理、ionic Native 开发实战、主题及样式的集成、CLI 命令行工具；最后通过通讯录例子简单讲解了 ionic 的具体开发实例。

通过本章内容的学习，希望读者能掌握以下这些技术点：

- 理解移动开发四种模式的技术特点，并根据场景进行技术选型。
- 理解 ionic 的技术架构，通过 Cordova 跟 Native 进行交互。

- 清楚 ionic 框架的各部分内容，能够利用 ionic 提供的服务进行开发。
- 了解安装和搭建 ionic 的开发环境，编写自己的应用。
- 掌握常用的 CLI 命令，提高开发效率。

与 Angular 一样，ionic 也处于不断迭代中，其主要内容基本已经稳定。开发者在使用该框架的同时，也请持续关注官方的 ChangeLog。只有拥抱变化才能紧跟技术发展的潮流，这是对技术开发者的挑战，也是互联网领域独特的魅力。

20

服务端渲染

20.1 概述

传统的服务端渲染是指 ASP、JSP 及 PHP 等后端渲染模式，即 Web 服务器从后台数据库获取数据，并把这些数据填充到 HTML 模版中，最终返回给浏览器的渲染机制。后来以 jQuery、Backbone、Angular、React 和 Vue 等为代表的前端技术框架兴起，客户端也可以承载复杂的业务场景，渲染的趋势逐渐从传统的服务端渲染转变为客户端渲染，而后端仅响应请求数据。

随着前端技术的不断发展，特别是 Node.js 的流行，又产生了一种新的渲染模式，即同构渲染（Isomorphic）。同构渲染强调一套代码能同时运行在服务端和客户端。当前常见的应用场景是在客户端渲染的基础上，把首屏变为类似于传统的服务端渲染，后续的逻辑交由客户端继续处理，它们的关系如图 20-1 所示。

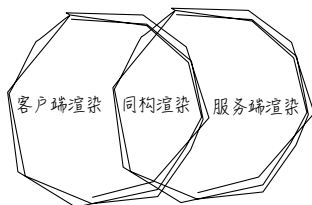


图 20-1 客户端渲染、同构渲染和服务端渲染的关系

随着对同构渲染的关注度越来越高，各大前端技术框架原先仅支持客户端渲染，近年来开始加入服务端渲染的支持，以达到同构渲染的效果。这里的服务端渲染跟传统的服务端渲染并不完全一样，新形式下的服务端渲染偏重复用原客户端渲染的业务逻辑代码，数据的拉取及其模板的绑定均在服务端完成，最终输出完整的 HTML 文档。

后文所述的服务端渲染均为新形式下的服务端渲染，并趋向于同构渲染。本章首先会介绍客户端渲染的局限性，引出服务端渲染；接下来介绍服务端的一些流程及场景局限性；最后通过通讯录实例讲述普通的客户端渲染例子如何改造成服务端渲染。

20.2 客户端渲染的局限性

客户端渲染首先加载必要的资源，加载完成后通常需要向服务端拉取数据，得到数据后重绘页面。在客户端渲染模式里，除应用数据需要与服务端通信外，用户主要的交互操作均在客户端独立完成。客户端渲染的流程如图 20-2 所示。

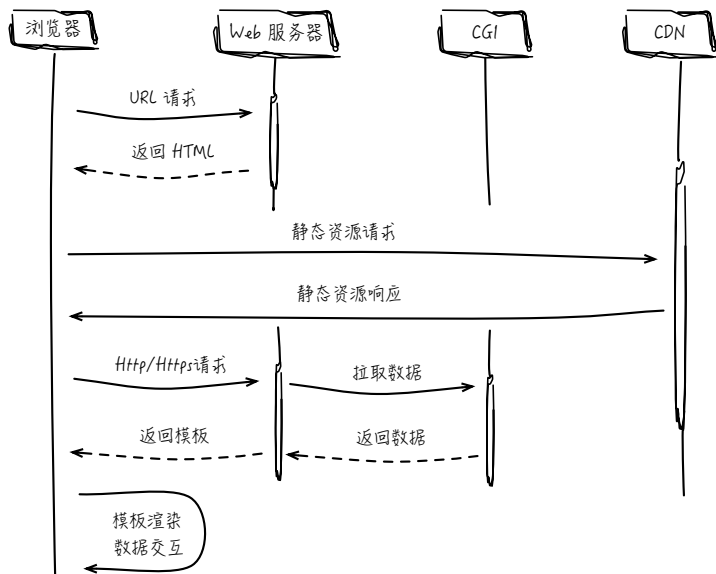


图 20-2 客户端渲染流程图

由客户端渲染的流程可以看出，当用户输入 URL 后，浏览器开始拉取静态 HTML 页面并解析文档对象，接着依次加载并解析文档中标记的 CSS 和 JavaScript 文件，并且向服务器发出 CGI 请求来拉取页面展示的数据，最后渲染到页面上。

这种前后端分离的客户端渲染模式已经成为 Web 应用开发的主流模式，虽然在局

部刷新、懒加载、富交互、CDN 部署及关注度分离等方面存在一定的优势，但也暴露了一些问题或缺陷。

首屏体验不佳

在呈现最终内容之前，页面需要加载数量多、体积大的资源文件，同时必须等待 JavaScript 加载完成后才会发起后端数据请求。这种强依赖的关系使得整个应用的首屏渲染耗时增加不少，从而影响了首屏体验。

SEO 不友好

搜索引擎爬虫对异步加载的数据处理得不够好，即使谷歌搜索引擎爬虫可以解析 AJAX 请求，但是网页原始数据源的权重远高于动态生成的内容，主流的前端框架无论 Angular、React 还是 Vue，它们构建的这些 SPA（单页应用）页面，在目前的搜索引擎处理方式下，基本无法满足 SEO 方面的需求。

20.3 服务端渲染的局限性

客户端渲染有其缺陷，但依然成为事实上的主流模式。尽管服务端渲染正好弥补了客户端渲染的劣势，但并没有成为主流模式，主要是因为服务端渲染也有其自身的局限性。

要了解其局限性，首先看看它的渲染流程，如图 20-3 所示。

- 浏览器根据用户行为发出 HTTP 请求。
- 服务端（指图 20-3 中的 Web 服务器）根据请求路径，初始化该路径对应的路由、组件等模块。
- CGI 程序根据业务逻辑处理数据，并返回给服务端。
- 服务端根据 CGI 的返回数据渲染模板，返回完整的页面。
- 客户端浏览器接收到服务端返回的完整 HTML 文档并渲染。
- 用户可以看到初始的应用界面。
- 浏览器发出请求获取页面中的 JavaScript 及其他静态资源（如 CSS、图片等）。
- JavaScript 加载就绪，客户端接管接下来的用户操作。

由客户端渲染和服务端渲染的流程可以看出其中最大的区别在于，使用服务端渲染的响应结果是已渲染的 HTML，这意味着浏览器将立足于服务器渲染的 HTML，而无须

等待全部 JavaScript 代码下载完成与执行。而客户端渲染一般是打开无数据填充的较为简单的 HTML，等待相关静态资源下载完成，执行数据请求并渲染后，才在浏览器中呈现出来。

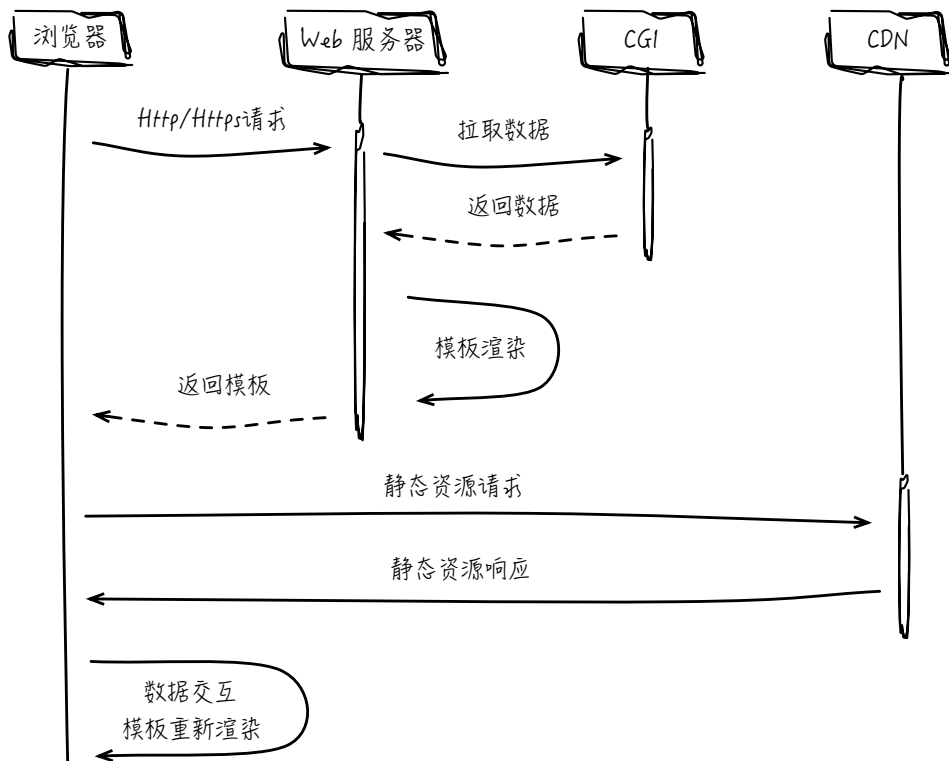


图 20-3 通用服务端渲染流程图

尽管服务端渲染解决了客户端渲染的一些问题，但由于渲染是在服务端完成的，容易出现性能问题。并且，服务端复用了客户端代码，除增加了项目的实施难度之外，还需要处理服务端和客户端环境差异的适配，例如异步数据拉取、环境差异、事件脱节等。

异步数据拉取

JavaScript 使用 XMLHttpRequest 对象异步请求服务端数据，并将数据更新到应用界面中。但在服务端渲染中，应用需要将当前页面的请求及相应的数据都准备好，并渲染输出 HTML 文本后再返回给浏览器。这在同构的服务端渲染下是一个挑战，因为服务端并没有 XMLHttpRequest 对象，这时就需要在服务端中替换该对象，或者更改代码以支持异步数据的获取，并且还要知道这些异步操作何时完成，以便将必要的数据一并渲

染后返回。另外，在浏览器中完成客户端视图渲染后，由于客户端代码同样会发起异步请求，这就导致本身在服务端渲染好的数据到了客户端还会再重新请求一次，增加了多余的请求。

环境差异

浏览器和服务端的环境差异，也是一个重要的问题。在实现服务端渲染之前，开发者在编写前端代码的时候，不会关注这些差异，比如 `window`、`document`、`navigator` 之类的对象在服务端是不存在的，如果在渲染逻辑中使用到这些对象，就会产生错误，而且在服务端操作 DOM 对象也是不允许的。

事件脱节

事件脱节发生在客户端已经展示了服务端渲染的界面，但客户端脚本尚未准备好的这个时间段里，这个时候用户在界面上的交互事件并不会得到预期的响应，状态数据也无法及时更新。想像一下，浏览器在接收到服务端渲染好的 HTML 文本后，开始绘制用户界面，这个时候同时会发出 JavaScript 等脚本的请求，由于网络环境或者浏览器的差异，JavaScript 请求发送、返回和执行的速度不同，用户界面可能已经就绪并且用户已经开始进行交互，如输入文本或点击按钮等操作，但此时脚本可能还没准备好，于是便造成了事件脱节，这可能会导致用户交互功能异常，如点击按钮没有反应等。

所以，除非真的需要通过服务端渲染带来的优势来提升项目的体验，否则并不建议广泛使用服务端渲染。例如一些内部的管理系统，对首屏渲染要求不高，也没有 SEO 的需求，则不必使用服务端渲染的架构。

20.4 Angular Universal 介绍

上述章节介绍了通用的服务端渲染原理，以及服务端渲染带来的问题，那么 Angular 是如何实现这套服务端渲染流程的，又是怎样解决服务端渲染出现的这些问题呢？接下来将详细介绍 Angular 的服务端渲染方案，即 Angular Universal。

Angular Universal 起初是一个社区项目，由 Jeff Whelpley 和 Patrick Stapleton 开发，后来被 Angular 4 集成，并迁移到 `@angular/platform-server` 模块中。

在 Angular Universal 的核心模块 `platformServer` 中，其内部常用的子模块有 `PlatformState`、`platformDynamicServer`、`platformServer`、`renderModule`、`renderModuleFactory`。

- `platformServer` 用于初始化要被渲染的 HTML 文档并根据 URL 指向特定的服务。
- `renderModuleFactory` 是 `platformServer` 内部提供的接口，用于更便捷地初始化状态量并将 HTML 文档解析到虚拟 DOM 树中，同时生成 DOM 状态序列化的字符串。
- `renderModule` 用于根据服务的业务逻辑，重新渲染 HTML 文档。
- `PlatformState` 通过 `renderToString()` 方法获取渲染的 HTML 字符串，通过 `getDOM()` 方法获取 DOM 树的结构等。



需要注意的是，`platformServer`、`renderModuleFactory` 在 AoT 模式下使用，对应的 JiT 模式下的版本为 `platformDynamicServer`、`renderModule`。

使用 Angular Universal 能快速地实现服务端渲染的基本流程，但要达到比较满意的使用体验，还需要解决前面章节中提到的一些服务端渲染的通病。

得益于 Zone.js，Angular Universal 能够感知整个异步任务的执行状态。在服务端渲染期间发出的数据请求，Angular Universal 可以知道数据何时能成功返回，当数据成功返回后可以继续进行渲染工作，渲染完成即可返回包含数据的完整 HTML 文档。

针对客户端与服务端环境差异的问题，在 Angular 中可以使用依赖注入来解决，根据代码的执行上下文，借助于依赖注入来动态传入对象。另外，对于 DOM 的操作，Angular 也提供了诸如 `ElementRef`、`Renderer` 等封装对象。但这并不能完美地涵盖所有场景，在实际开发中仍需要开发者具备这种环境差异的意识。

同时为了解决事件脱节的问题，Angular Universal 提供了 Preboot 来处理，不需要开发人员干预，在后面的例子中将会说明 Preboot 如何使用。



Preboot 工具是可选的，并非在 Angular 服务端渲染的所有场景中都需要使用。

通过前面对服务端渲染的介绍，我们了解了服务端渲染的优势，并理解了其渲染流程、原理和问题，以及 Angular Universal 的解决方案，接下来将通过实例来详细介绍 Angular Universal。

20.5 将通讯录例子改造成 Angular Universal 的方式

下面将使用本书中的通讯录例子，讲述如何从一个 Angular CLI 生成的纯前端项目，一步步配置成 Angular Universal 项目。



生成 Angular 初始项目的步骤请参照第 4 章的相关章节，这个项目基于 Angular CLI 打造，Angular CLI 本身一直在迭代更新，一些用法可能会发生变化，遇到有差异的地方建议读者翻查 ChangeLog 来了解。

安装依赖

由于在 Angular 4 版本之后，Angular Universal 被迁移到了 @angular/platform-server 下，而 @angular/platform-server 又依赖 @angular/animations 这个模块，所以首先需要安装它们。

```
npm install --save @angular/platform-server @angular/animations
```

修改 angular-cli.json 配置文件

在目前的 .angular-cli.json 配置文件中，可以看到 apps 下的配置，其中 main 和 tsconfig 分别指客户端渲染应用的入口文件和编译配置文件，要使应用支持服务端渲染，需要增加相应的配置。增加的代码如下：

```
// .angular-cli.json

...
"apps": [ // 增加 ssr 应用配置
  ...
  {
    "name": "ssr",
    "root": "src",
    "outDir": "dist-ssr",
    "assets": [
      "assets",
      "favicon.ico"
    ],
    "index": "index.html",
    "main": "main.server.ts",
```

```
"test": "test.ts",
"tsconfig": "tsconfig.server.json",
"prefix": "app",
"styles": [
  "styles.css"
],
"scripts": [],
"environmentSource": "environments/environment.ts",
"environments": {
  "dev": "environments/environment.ts",
  "prod": "environments/environment.prod.ts"
},
"platform": "server"
}
],
// ...
```

可以看到，它和客户端渲染的配置差不多，不同的是以下几点。

- 增加了 `name` 配置，用来在命名上区分原来的配置，此处命名为“`ssr`”。
- 删除了 `polyfills` 和 `testTsconfig` 配置项。
- `outDir`: 表示运行脚本的输出目录，此处为了和客户端渲染区分开，使用了 `dist-ssr` 目录。
- `main`: 原来的入口文件并不适用于服务端渲染，因为它调用了 `platformBrowser`，为此增加了一个 `main.server.ts` 用于服务端渲染，后面会有进一步讲述。
- `tsconfig`: 为了区分客户端的编译配置，增加了一个 `tsconfig.server.json` 文件，后面也会讲述。
- 声明 `platform` 为 `server`。

创建服务端渲染根模块

客户端渲染创建了 `AppModule` 作为应用的入口模块，即根模块。而对于服务端渲染，也需要创建一个入口模块，这里命名为 `AppServerModule`，并且导入客户端入口模块 `AppModule` 以复用原有逻辑。示例代码如下：

```
// app.server.module.ts

import { NgModule } from '@angular/core';
```



```
import { ServerModule } from '@angular/platform-server';
import { AppModule } from './app.module';
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    AppModule, // 导入客户端根模块
    ServerModule
  ],
  bootstrap: [AppComponent],
  providers: []
})
export class AppServerModule {}
```

代码内容很简单，引入 `@angular/platform-server` 和复用 `AppModule` 并作为 `imports` 属性导入即可，同样也指定 `AppComponent` 作为启动组件。

接下来，在客户端的根模块里，需要对原来的 `BrowserModule` 稍作修改，使用 `withServerTransition()` 方法并导入到模块中。该方法需要一个 `appId`，这里设置为 `angular-contacts-demo`。示例代码如下：

```
// app.module.ts

// ...
@NgModule({
  // ...
  imports: [
    // 将 BrowserModule 替换为下面的形式
    BrowserModule.withServerTransition({
      appId: 'angular-contacts-demo'
    }),
    // ...
  ],
  // ...
})
export class AppModule { }
```

创建服务端渲染入口文件

在上文提到的 `.angular-cli.json` 文件中配置服务端入口文件，命名为 `main.server.ts`，其内容很简单，导出服务端根模块即可。示例代码如下：

```
// main.server.ts

export { AppServerModule } from './app/app.server.module';
```

创建服务端编译配置

该文件即上文提到的 `tsconfig.server.json` 文件，复制 `tsconfig.app.json` 为 `tsconfig.server.json`，增加 `angularCompilerOptions`，并修改 `module` 配置项为 `"commonjs"`。示例代码如下：

```
// tsconfig.server.json

{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "outDir": "../out-tsc/app",
    "module": "commonjs",
    "baseUrl": "",
    "types": []
  },
  "exclude": [
    "test.ts",
    "**/*.spec.ts",
    "testing/**"
  ],
  "angularCompilerOptions": {
    "entryModule": "app/app.server.module#AppServerModule"
  }
}
```

创建 Web Server

上述只是完成了配置，要把服务端渲染运行起来，还需要创建一个 HTTP 服务，把 `dist-ssr` 引入并作为后端渲染服务所需的脚本等资源，并且把 `dist` 目录作为静态资源服务。示例代码如下：

```
// server-aot.js

require('zone.js/dist/zone-node');
const fs = require('fs');
const path = require('path');
const http = require('http');
const Static = require('node-static');
const files = new Static.Server(path.join(__dirname, 'dist'));

const { AppServerModuleNgFactory } = require('./dist-ssr/main.bundle');
const { renderModuleFactory } = require('@angular/platform-server');

http.createServer((req, res) => {
  const path = req.url;
  if(
    path.indexOf('/assets') === 0 ||
    path === '/favicon.ico' ||
    /\.js(?:\.map)?$/\.test(path)
  ) {
    files.serve(req, res);
  } else {
    renderModuleFactory(AppServerModuleNgFactory, {
      url: path,
      document: fs.readFileSync('dist/index.html', 'utf8')
    }).then(html => {
      res.end(html);
    });
  }
}).listen(4200);

console.log('open browser for http://localhost:4200');
```

代码中引入了 zone.js（注意跟浏览器 polyfills 里引入的 zone 不是同一个），因为 Angular 需要使用 Zone.js 的特性来跟踪代码中的异步调用，并在异步调用结束后响应请求并渲染 HTML 文本。另外，为了方便使用，代码中使用了 node-static 为静态资源提供服务，可以通过 `npm install --save node-static` 安装。

需要说明的是，在 Angular 的后端渲染服务中，由于 HTTP 请求不允许使用相对路径，所以在改造服务端渲染时还需要把 `contact.service.ts` 中的请求路径改成完整的

HTTP 路径。示例代码如下：

```
// contact.service.ts

// ...
// const CONTACT_URL = '/assets/contacts.json';
const CONTACT_URL = 'http://localhost:4200/assets/contacts.json';
// ...
```



在实际项目中并不会把请求路径写成绝对路径，通常客户端请求都会是本域的请求，所以用相对路径是没有问题的。但实际的数据可能会分布在不同域名的服务上，一般的做法是在提供静态资源的服务上使用 Nginx 等服务代理进行接口转发处理。不过这已超出本书的讨论范围，为了简便起见，此处使用了绝对路径。

增加构建命令并启动服务

至此，服务端渲染准备工作就完成了。接下来在 `package.json` 里增加以下构建脚本，用于构建最终的运行代码。示例代码如下：

```
// package.json

// ...
"scripts": {
  // ...
  "build-ssr": "ng build && ng build --app ssr",
  "build-ssr:aot": "ng build --aot && ng build --aot --app ssr",
  "start-ssr": "npm run build-ssr && node server",
  "start-ssr:aot": "npm run build-ssr:aot && node server-aot"
}
// ...
```

命令根据编译模式分为两部分：JiT 编译（不带 `--aot` 参数）和 AoT 编译（带 `--aot` 参数）。其中 `ng build --app ssr` 表示使用前面在 `.angular-cli.json` 文件中增加的编译项来生成服务端渲染的文件。



如果不添加 `--aot` 参数，则默认使用 JiT 方式编译，这种方式一般用在开发环境中，是为了调试方便。读者可以先运行 `npm start` 看看客户端渲染返回的 HTML，以便和服务端渲染进行对比。

运行 `npm run start-ssr:aot` 命令，编译成功后，可以看到在项目目录下同时输出了 `dist` 和 `dist-ssr` 目录。服务启动后，在 Chrome 浏览器中访问 `http://localhost:4200`，并打开开发者工具，可以看到 Network 这个 Tab 中的 document 请求，返回的 HTML 已经是完整的内容了，对比客户端渲染版本返回的 HTML 代码可以看到变化。至此，我们就成功地将一个客户端渲染项目，配置成支持 Angular Universal 服务端渲染的项目。

客户端渲染示例代码如下：

```
<!-- Network 中返回的 index.html 请求内容 -->

<!DOCTYPE html>
<html>
<!-- ... -->
<body>
  <!-- 在客户端渲染中只有一个 app-root -->
  <app-root>
    Loading...
  </app-root>

  <!-- Angular 插入的脚本 -->
  <script type="text/javascript" src="inline.bundle.js"></script>
  <script type="text/javascript" src="polyfills.bundle.js"></script>
  <script type="text/javascript" src="styles.bundle.js"></script>
  <script type="text/javascript" src="vendor.bundle.js"></script>
  <script type="text/javascript" src="main.bundle.js"></script>
</body>
</html>
```

服务端渲染示例代码如下：

```
<!-- Network 中返回的 index.html 请求内容 -->

<!DOCTYPE html>
<html>
<!-- ... -->
```

```
<body>
  <!-- 在服务端渲染中把渲染好的 HTML 返回 -->
  <app-root _ngghost-c0="" ng-version="4.3.1">
    <main _ngcontent-c0="" class="main">
      <router-outlet _ngcontent-c0=""></router-outlet>
      <app-list _ngghost-c1="">
        <!-- ... -->
      </app-list>
    </main>
  </app-root>
  <!-- Angular 插入的脚本 -->
  <script type="text/javascript" src="inline.bundle.js"></script>
  <script type="text/javascript" src="polyfills.bundle.js"></script>
  <script type="text/javascript" src="styles.bundle.js"></script>
  <script type="text/javascript" src="vendor.bundle.js"></script>
  <script type="text/javascript" src="main.bundle.js"></script>
</body>
</html>
```

20.6 服务端渲染的进阶实践

前面通过通讯录例子讲述了如何将 Angular 客户端渲染应用改造成支持服务端渲染的 Angular Universal 应用，这是最基础的步骤，现在来进一步看看更多的实践，也就是前面提到过的服务端渲染可能带来的问题，即异步数据拉取、环境差异和事件脱节。

对于异步数据拉取，我们知道 Angular 使用了 Zone 的特性，使得在服务端渲染时，能跟踪异步事件并等待它们完成后再将数据和 HTML 文件一并渲染，然后返回给浏览器。这也导致了一个小问题，浏览器解析了服务端渲染好的 HTML 并下载了必要的脚本且启动应用后，客户端会再次向服务器请求相同的数据，这个时候会造成重复的数据请求，并且请求完成后可能会造成页面闪烁，导致不友好的用户体验。针对这个问题，解决方案是把服务端的数据“同步”给客户端，避免重复请求。

对于环境差异和事件脱节，前面的内容已经提到了可以分别使用依赖注入和 Pre-boot 来解决。

下面将分别通过示例来讲解。

20.6.1 服务端数据的同步

如何将服务端的数据同步到客户端呢？可以通过内嵌脚本的方式，将数据序列化并渲染到 HTML 中一并返回，然后客户端在解析 HTML 的时候，就会执行到这个脚本，把数据反序列化并存储到 window 对象中，当应用启动后，异步请求会先检测这个对象，而不是直接向服务端请求数据。

假设应用有一个 /requestUrl 这样的请求，并返回数据 “HelloWorld”，那么在服务端渲染时，在 HTML 中插入类似于如下的脚本：

```
<!-- ... -->
<head>
  <script>window['/requestUrl'] = "HelloWorld"</script>
</head>
<!-- ... -->
```

当 HTML 返回给客户端解析后，该脚本就会被执行，“HelloWorld” 的值会被写入 window 对象中，异步请求就可以直接使用该数据了。

Angular 提供了 ServerTransferStateModule 和 BrowserTransferStateModule 模块，方便开发者实现这一功能。接下来通过通讯录例子来实现数据的同步，主要步骤如下：

- 分别在 app.server.module.ts 和 app.browser.module.ts 中引入上述模块。
- 修改 ContactService 类，优先获取缓存中同步的数据。
- 在 main.ts 中对应用启动稍作修改，确保应用在 DOM 准备好后再启动。

引入相关模块

分别在 app.server.module.ts 和 app.browser.module.ts 中引入 ServerTransferStateModule 和 BrowserTransferStateModule 模块，示例代码如下：

```
// app.server.module.ts

// ...
import { ServerModule, ServerTransferStateModule } from '@angular/platform-server';

@NgModule({
  imports: [
    // ...
    ServerTransferStateModule
  ],
```

```
// ...
})
export class AppServerModule {}

// app.browser.module.ts

// ...
import { BrowserTransferStateModule } from '@angular/platform-browser';

@NgModule({
  imports: [
    // ...
    BrowserTransferStateModule
  ],
  // ...
})
export class AppBrowserModule {}
```

修改 ContactService 类

Angular 提供了 `TransferState` 来处理数据的同步逻辑，通过它的 `get()`、`set()` 方法来分别获取、设置同步的数据，这两个方法需要接收一个 `StateKey` 对象参数，Angular 提供了 `makeStateKey()` 方法来创建。示例代码如下：

```
// contact.service.ts

// ...
// 导入 TransferState 和 makeStateKey
import { TransferState, makeStateKey } from '@angular/platform-browser';
// ...

// 生成缓存key值
const CONTACTS_KEY = makeStateKey('contacts');
// ...

@Injectable()
export class ContactService {
  constructor(private http: HttpClient, private state: TransferState) {}
  // ...
}
```


接着修改 `getContactsData()` 方法，增加 `state.set()` 和 `state.get()` 相关逻辑。示例代码如下：

```
// contact.service.ts

// ...
getContactsData(opts?: any) {
  let source;
  let cache = this.state.get(CONTACTS_KEY, null as any); // 客户端获取数据
  if (cache) {
    source = Observable.of(cache);
  } else {
    source = this.http.get(CONTACT_URL)
      .do(data => {
        this.state.set(CONTACTS_KEY, data as any); // 服务端缓存数据
      })
      .catch(this.handleError);
  }
  return source.map(data => this.filter(data, opts));
}
// ...
```



`state.get()` 方法的第二个参数是指定默认的返回数据，即不存在 `CONTACTS_KEY` 对应的数据时返回该值。

`getContactsData()` 方法在服务端运行期间，`state` 里并没有存储数据，因此服务端会发出请求获取数据，数据返回后通过 `state.set()` 方法将数据缓存起来。服务端执行结束时，会将 `state` 里的数据序列化到 HTML 一并发送至客户端。客户端启动运行时，首先从 HTML 中读取服务端发送过来的数据，并反序列化还原成数据对象，然后当客户端执行 `getContactsData()` 时，通过 `state.get()` 即可获得到相关数据。

这正是 `TransferState` 的巧妙之处，通过依赖注入分别在服务端和客户端进行了不同的实现。在服务端，`TransferState` 的作用是序列化数据；而在客户端，`TransferState` 的作用是反序列化数据，从而避免了重复的数据请求。

修改 main.ts

为了确保 Angular 能正确地将数据从 HTML 中反序列化，需要确保 HTML 文档已渲染完成，因此将启动代码修改为在 DOMContentLoaded 后调用。示例代码如下：

```
// main.ts

// ...
document.addEventListener('DOMContentLoaded', () => {
  platformBrowserDynamic().bootstrapModule(AppBrowserModule)
    .catch(err => console.error(err));
});
```

在控制台运行 `npm run start-ssr` 后，在浏览器中访问联系人页面，浏览器已经不再发出 `contacts.json` 的请求，并且在后端返回的 HTML 文本的底部可以看到 `<script id="angular-contacts-demo-state" type="application/json">...` 这样的代码，这便是 Angular 在服务端序列化的数据并同步到了浏览器端。

20.6.2 使用依赖注入解决环境差异

大家知道，客户端和服务端的环境并不完全一致，对于一些客户端专有功能，如 `sessionStorage`，可以在服务端新建一个类似的服务，并统一使用依赖注入的方式“抹平”调用的差异。

假设通讯录例子新增了一个功能，通过 `sessionStorage` 来持久化联系人数据，避免在刷新页面的时候重新请求。对于纯前端应用来说，通常的做法是直接调用 `sessionStorage` 的 `setItem()` 和 `getItem()` 方法。示例代码如下：

```
// contact.service.ts

// ...

getContactsData(opts?: any) {
  let source;
  // 如果 sessionStorage 中存在缓存，则获取缓存中的数据
  // 该代码在服务端运行时出错，因为服务端没有 sessionStorage 对象
  const contacts = sessionStorage.getItem('contacts');
  if (Array.isArray(contacts)) {
    source = Observable.of(contacts);
  } else {
    source = this.http.request('get', CONTACT_URL)
```

```
        .do(data => sessionStorage.setItem('contacts', data))
        .catch(this.handleError);
    }
    return source.map(data => this.filter(data, opts));
}
// ...
```

上述代码在服务端的环境中运行会出错，针对环境差异这个问题，在 Angular 中便可以使用依赖注入来解决。

首先创建一个自定义的 SessionStorage 类。示例代码如下：

// 该代码仅仅是示例，并没有提供源码

```
import { Injectable, PLATFORM_ID, Inject } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';

@Injectable()
export class SessionStorage {
    private isBrowser: boolean = isPlatformBrowser(this.platformId);
    private serverStorage = {};
    constructor(@Inject(PLATFORM_ID) private platformId) {}

    public setItem(key: string, value: any): void {
        if (this.isBrowser) {
            window.sessionStorage.setItem(key, value);
        } else {
            this.serverStorage[key] = value;
        }
    }

    public getItem(key: string): any {
        if (this.isBrowser) {
            return window.sessionStorage.getItem(key);
        } else {
            return this.serverStorage[key];
        }
    }
}
```

在自定义的 `SessionStorage` 类中，判断平台是浏览器的时候才调用相应的方法，接着把这个对象注入到 `ContactService` 类中即可。示例代码如下：

// 该代码仅仅是示例，并没有提供源码

// ...

```
import { SessionStorage } from './session-storage';

export class ContactService {
  constructor(private http: HttpClient, private sessionStorage: SessionStorage) {}

  getContactsData(opts?: any) {
    let source;
    let cache = this.sessionStorage.getItem('contacts');
    if (Array.isArray(cache)) {
      source = Observable.of(cache);
    } else {
      source = this.http.request('get', CONTACT_URL)
        .do(data => this.sessionStorage.setItem('contacts', JSON.stringify(data)))
        .catch(this.handleError);
    }
    return source.map(data => this.filter(data, opts));
  }
}
```



在使用 `SessionStorage` 前别忘了先在模块里注入。

20.6.3 使用 Preboot 解决事件脱节

Preboot 是一个可选的工具，它可以在服务端视图渲染完成后，记录用户交互的事件，例如用户的点击、文本的输入等，等到客户端视图渲染完毕后，将用户的交互“重放”一遍，这在一定程度上降低了事件脱节带来的交互迟缓。

使用 Preboot 的流程

使用 Preboot 的流程如图 20-4 所示。

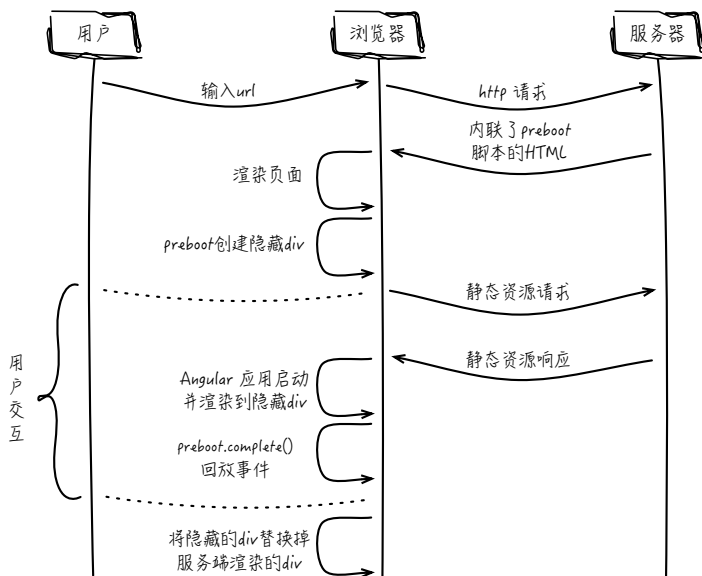


图 20-4 使用 Preboot 的流程图

- 用户输入 URL 后，浏览器发出 HTTP 请求。
- 服务端生成一个完整的 HTML 文档，并将 Preboot JavaScript 脚本内联在该 HTML 文档里。
- 浏览器接收到这个 HTML 文档并渲染页面。
- Preboot 在页面中创建了一个隐藏的 div 作为客户端视图渲染的容器。
- 这时候用户已经可以看到界面效果了，但这个由服务端渲染出来的界面并没有响应交互事件的能力，用户此时与页面进行交互都会被 Preboot 记录下来。
- 同时，浏览器发出静态资源的请求（如 JavaScript、CSS、图片等）。
- 当脚本就绪后，客户端的 Angular 应用启动，并渲染到 Preboot 创建的隐藏的 div 上。
- 当客户端渲染完成后，Preboot 会在隐藏的 div 上回放在客户端 Angular 应用启动之前的用户交互事件。
- Preboot 将隐藏的 div 显示并替换由服务端渲染的 div。
- 最后 Preboot 还会做一些清理工作，如重置输入焦点等。

通讯录例子中 Preboot 的使用

首先，通过 `npm install preboot --save` 命令将 Preboot 安装到项目中。

Preboot 支持直接导入到 Angular 模块中。在服务端渲染的模块中，只需要引入 `ServerPrebootModule`，并指定需要记录事件的节点即可。此示例中指定的是根节点 `app-root`。示例代码如下：

```
// app.server.module.ts

// ...
import { ServerPrebootModule } from 'preboot/server';
// ...

@NgModule({
  imports: [
    // ...
    ServerPrebootModule.recordEvents({ appRoot: 'app-root' })
  ],
  // ...
})
export class AppServerModule {}
```

在客户端的模块中，需要导入 `BrowserPrebootModule`。示例代码如下：

```
// app.browser.module.ts

import { NgModule } from '@angular/core';
import { BrowserPrebootModule } from 'preboot/browser';
import { AppModule } from './app.module';
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    AppModule,
    BrowserPrebootModule.replayEvents() // 在应用启动后重放事件
  ],
  bootstrap: [AppComponent],
  providers: []
})
export class AppBrowserModule {}
```

为了演示方便，下面将 `main.ts` 中的应用启动延迟 3 秒，可以验证 Preboot 在服务端渲染后记录事件并在客户端启动后重放的过程。示例代码如下：

```
// main.ts

// ...

setTimeout(() => {
  platformBrowserDynamic().bootstrapModule(AppBrowserModule)
    .catch(err => console.error(err));
}, 3000)
```

运行 `npm run start-ssr` 启动应用，在浏览器中打开 `http://localhost:4200/list`，看到界面后点击左下角的“收藏”按钮，此时会看到界面弹出一个蒙层，因为 Preboot 会对 `<button>` 标签的点击事件进行阻塞（这是 Preboot 默认的设置）。在大约 3 秒钟之后，可以看到蒙层自动消失并且路由跳转到了 `/collection` 页面，这说明 Preboot 成功录制并重放了该点击事件。

20.7 小结

本章首先介绍了传统的服务端渲染、客户端渲染和同构渲染的概念，并说明了客户端渲染的优缺点；然后重点讲述了服务端渲染原理，以及服务端渲染可能会带来异步数据拉取、环境差异及事件脱节等问题；接下来介绍了 Angular 的服务端渲染，即 Angular Universal 的解决方案。最后通过通讯录例子讲述了如何一步步将 Angular 项目改造成 Angular Universal 项目，并进一步讲解了如何在 Angular Universal 中解决服务端渲染带来的各种问题。